In The United States Patent and Trademark Office
On Appeal From The Examiner To The Board
of Patent Appeals and Interferences

In re Application of:	Richard H. Harvey
Serial No.:	09/721,806
Filed:	November 24, 2000
Art Unit.:	2164
Confirmation No.:	3613
Examiner:	Sathyanaraya R. Pannala
Title:	*A METHOD AND APPARATUS FOR OPERATING A DATABASE*

MAIL STOP APPEAL BRIEF - PATENTS
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Dear Sir:

Appeal Brief

Appellant has appealed to the Board of Patent Appeals and Interferences ("Board") from the decision of the Examiner dated March 31, 2008, finally rejecting Claims 1-5, 9, 10, 14-18, 22, 23, and 27-31. Appellant filed a Notice of Appeal on June 30, 2008. Appellant respectfully submits this Appeal Brief.

Table of Contents

Page

Table of Contents 2

Real Party In Interest 3

Related Appeals and Interferences 4

Status of Claims 5

Status of Amendments 6

Summary of Claimed Subject Matter 7

Grounds of Rejection to be Reviewed on Appeal 14

Argument 15

I. Claims 1, 9, 14, 22, and 31 satisfy the requirements of 35 U.S.C. § 112, first paragraph. 15

II. Claims 1, 9, 14, 22, and 31 satisfy the requirements of 35 U.S.C. § 112, second paragraph. 16

III. Claims 1-5, 9-10, 14-18, 22-23, and 27-31 satisfy the requirements of 35 U.S.C. § 101. 18

IV. Claims 1-5, 9-10, 14-18, 22-23, and 27-31 are allowable over the cited references. 21

Appendix A: Claims on Appeal 28

Appendix B: Evidence 38

Appendix C: Related Proceedings 39

Real Party In Interest

The inventor assigned this application to Computer Associates Think, Inc., pursuant to an assignment recorded at reel 011925, frame 0775. Accordingly, the real party in interest is Computer Associates Think, Inc.

Related Appeals and Interferences

Appellant, the undersigned Attorney for Appellant, and the Assignee know of no applications on appeal that may directly affect or be directly affected by or have a bearing on the Board's decision in the pending appeal.

Status of Claims

Claims 1-5, 9, 10, 14-18, 22, 23, and 27-31 were finally rejected by the decision of the Examiner dated March 31, 2008. Claims 6-8, 11-13, 19-21, and 24-26 have been withdrawn. Appellant presents Claims 1-5, 9, 10, 14-18, 22, 23, and 27-31 for appeal and set forth these claims in Appendix A.

Status of Amendments

All amendments submitted by Appellant were entered by the Examiner prior to the mailing of the Final Office Action dated March 31, 2008.

Summary of Claimed Subject Matter

A database system may include an apparatus for processing a directory service query such as, for example, a X.500 or LDAP service query. (P. 1, ll. 4-8; p. 6, ll. 14-22). The system may represent the query as a logic expression. (P. 6, ll. 14-22; Fig. 2). The system may expand the logic expression into a sum of terms. *Id.* For example, a query that is expressed as A.(B+C.!D) may be expanded into the following sum of terms: A.B + A.C.!D. Expanding the expression into a sum of terms may permit the system to optimize the processing of the query. (P. 8, ll. 1-2). For example, expanding the expression into a sum of terms may provide an opportunity to flatten terms, to evaluate terms in any order, to evaluate terms in parallel, or to stop processing based on a size or time limit. (P. 8, ll. 1-7).

To obtain the sum of terms, the system may remove nested instructions in order to simplify the expression representing the service query. (P. 8, ll. 15-17). Each term may then be converted into a flattened query. (P. 8, ll. 17-22). Thus, a relatively long expression can be converted into a number of smaller instructions (or expressions), each of which can be flattened. *Id.* The flattened and smaller instructions (or expressions) may run much faster than complex queries, thus improving performance of the system. *Id.*

Once the expression is expanded into a sum of terms, the system may determine whether a particular term includes a NOT operator. (P. 6, ll. 23-27). If the term does not include a NOT operator, the system may convert the term to a SQL instruction. (P. 6, ll. 26-27; Fig. 2a). However, if the term does include a NOT operator, then the term may be expanded into positive and/or negative terms. (P. 7, ll. 1-3). Thus, the query may be rewritten as an expression without NOT operators. (P. 9, ll. 15-26). For example, the term A.!B can be expressed as A.(1-B), which can be written as A - AB. *Id.* This expression no longer includes the NOT operator. *Id.* If the system supports subtraction, then the system may collect all positive terms in a list, collect all negative terms into another list, and then subtract the positive term list and the negative term list whilst ignoring duplicates. (P. 9, l. 29 – p. 10, l. 2). If the system does not support subtraction, then the system may collect all negative terms in a list and, in the process of collecting all positive terms in another list, only keep the terms that are not in the negative list. (P. 10, ll. 3-5).

In some embodiments, the system may process a service query with higher order subtractions. (P. 10, ll. 7-9). For example, the system may process a query of A.!B.!C into the following sum of terms: A - A.C - A.B +A.B.C. (P. 10, ll. 10-15). This expression can

be further processed (or evaluated) to remove or ignore duplicate or overlapping results. (P. 10, ll. 16-25). The A operation may provide a positive list, the A.C operation may provide a negative list, and the A.B operation may also provide a negative list. *Id.* The lists may be evaluated in any order, or in parallel, or in accordance with the optimizations noted above. *Id.* By ignoring duplicate results, there may be no need to evaluate the term A.B.C. *Id.* With reference to mathematical principals, the subtraction of A.B and A.C may effectively subtract A.B.C. twice, which is a duplicate result. *Id.* The system described above may be applied to the evaluation / execution of database service queries. (P. 11, ll. 12-13).

For the convenience of the Board, Appellant provides the following mapping of the independent claims here on appeal. Appellant does not necessarily identify all portions of the specification and drawings relevant to the recited elements of the claims. Appellant provides the following mapping to help the Board make a decision on this Appeal, not to limit the scope of the claims.

Claim 1

A method of processing a database service query, comprising: (*see, e.g.,* p. 6, ll. 14-22; p. 8, ll. 26 - p. 9, l. 2; Fig. 2)

receiving a service query, (*see, e.g.,* p. 6, ll. 14-22; Fig. 2)

obtaining a sum of terms associated with the service query by expanding at least one nested term into one or more un-nested terms, (*see, e.g.,* p. 6, ll. 14-22; p. 7, ll. 24-29; p. 8, ll. 1-24; p. 9, l. 15 - p. 10, l. 25; Figs. 2, 2a, 2b)

evaluating the sum of terms as a plurality of SQL instructions, wherein the sum of terms comprises one or more positive terms and one or more negative terms, (*see, e.g.,* p. 6, ll. 23-30; p. 7, ll. 1-10; p. 8, ll. 8-29; Figs. 2, 2a, 2b)

determining a plurality of results associated with the sum of terms, wherein the plurality of results are determined by a processor and the determination comprises: (*see, e.g.,* p. 1, ll. 1-8; p. 6, ll. 14-22; p. 10, ll. 3-25; Figs. 2, 2a, 2b)

collecting, into a first list, results associated with the one or more negative terms, and (*see, e.g.,* p. 10, ll. 3-6; p. 10, ll. 16-22)

collecting, into a second list, results associated with the one or more positive terms while omitting from the second list any results that are in the first list, (*see, e.g.,* p. 10, ll. 3-6; p. 10, ll. 16-22)

and

providing one or more results from the second list to a user. (*see, e.g.,* p. 1, ll. 15-22)

Claim 9

A system for processing a directory service query, comprising: (*see, e.g.,* p. 1, ll. 1-8; p. 6, ll. 14-22; p. 8, ll. 26 - p. 9, l. 2; Fig. 2)

a database operable to store arbitrary data; and (*see, e.g.,* p. 1, ll. 1-8; p. 8, ll. 14-24)

a processor that is communicatively coupled to the database and that processes a service query by: (*see, e.g.,* p. 6, ll. 14-22; p. 1, ll. 1-8)

obtaining a sum of terms by expanding at least one nested term into one or more un-nested terms, (*see, e.g.,* p. 6, ll. 14-22; p. 7, ll. 24-29; p. 8, ll. 1-24; p. 9, l. 15 - p. 10, l. 25; Figs. 2, 2a, 2b)

evaluating the sum of terms as a plurality of SQL instructions, wherein the sum of terms comprises one or more positive terms and one or more negative terms, (*see, e.g.,* p. 6, ll. 23-30; p. 7, ll. 1-10; p. 8, ll. 8-29; Figs. 2, 2a, 2b)

determining a plurality of results associated with the sum of terms, wherein the determination comprises: (*see, e.g.,* p. 6, ll. 14-22; p. 10, ll. 3-25; Figs. 2, 2a, 2b)

collecting, into a first list, results associated with the one or more negative terms, and (*see, e.g.,* p. 10, ll. 3-6; p. 10, ll. 16-22)

collecting, into a second list, results associated with the one or more positive terms while omitting from the second list any results that are in the first list, (*see, e.g.,* p. 10, ll. 3-6; p. 10, ll. 16-22)

and

providing one or more results from the second list to a user. (*see, e.g.,* p. 1, ll. 15-22)

Claim 14

A method of processing a directory service query, comprising: (*see, e.g.,* p. 6, ll. 14-22; p. 8, ll. 26 - p. 9, l. 2; Fig. 2)

receiving a directory service query, (*see, e.g.,* p. 6, ll. 14-22; Fig. 2)

obtaining a sum of terms by expanding at least one nested term into one or more un-nested terms, (*see, e.g.,* p. 6, ll. 14-22; p. 7, ll. 24-29; p. 8, ll. 1-24; p. 9, l. 15 - p. 10, l. 25; Figs. 2, 2a, 2b)

mapping the sum of terms to a plurality of SQL instructions, wherein the sum of terms comprises one or more positive terms and one or more negative terms, (*see, e.g.,* p. 6, ll. 23-30; p. 7, ll. 1-10; p. 8, ll. 8-29; Figs. 2, 2a, 2b)

determining a plurality of results associated with the sum of terms, wherein the plurality of results are determined by a processor and the determination comprises: (*see, e.g.,* p. 1, ll. 1-8; p. 6, ll. 14-22; p. 10, ll. 3-25; Figs. 2, 2a, 2b)

collecting, into a first list, results associated with the one or more negative terms, and (*see, e.g.,* p. 10, ll. 3-6; p. 10, ll. 16-22)

collecting, into a second list, results associated with the one or more positive terms while omitting from the second list any results that are in the first list, (*see, e.g.,* p. 10, ll. 3-6; p. 10, ll. 16-22)

and

providing one or more results from the second list to a user. (*see, e.g.,* p. 1, ll. 15-22)

Claim 22

A system for processing a directory service query, comprising: (*see, e.g.,* p. 1, ll. 1-8; p. 6, ll. 14-22; p. 8, ll. 26 - p. 9, l. 2; Fig. 2)

a database that is operable to store arbitrary data; and (*see, e.g.,* p. 1, ll. 1-8; p. 8, ll. 14-24)

a processor that is communicatively coupled to the database and that processes a directory service query by: (*see, e.g.,* p. 6, ll. 14-22; p. 1, ll. 1-8)

obtaining a sum of terms by expanding at least one nested term into one or more un-nested terms, (*see, e.g.,* p. 6, ll. 14-22; p. 7, ll. 24-29; p. 8, ll. 1-24; p. 9, l. 15 - p. 10, l. 25; Figs. 2, 2a, 2b)

mapping the sum of terms to a plurality of SQL instructions, wherein the sum of terms comprises one or more positive terms and one or more negative terms, (*see, e.g.,* p. 6, ll. 23-30; p. 7, ll. 1-10; p. 8, ll. 8-29; Figs. 2, 2a, 2b)

determining a plurality of results associated with the sum of terms, wherein the determination comprises: (*see, e.g.,* p. 6, ll. 14-22; p. 10, ll. 3-25; Figs. 2, 2a, 2b)

collecting, into a first list, results associated with the one or more negative terms, and (*see, e.g.,* p. 10, ll. 3-6; p. 10, ll. 16-22)

collecting, into a second list, results associated with the one or more positive terms while omitting from the second list any results that are in the first list, (*see, e.g.,* p. 10, ll. 3-6; p. 10, ll. 16-22)

and

providing the determined plurality of results to a user. (*see, e.g.,* p. 1, ll. 15-22)

Claim 31

A method of processing a database service query, comprising: (*see, e.g.,* p. 6, ll. 14-22; p. 8, ll. 26 - p. 9, l. 2; Fig. 2)

receiving a service query; (*see, e.g.,* p. 6, ll. 14-22; Fig. 2)

obtaining a sum of terms associated with the service query by: (*see, e.g.,* p. 6, ll. 14-22; p. 7, ll. 24-29; Fig. 2)

expanding at least one nested term into one or more un-nested terms; (*see, e.g.,* p. 6, ll. 14-22; p. 7, ll. 24-29; p. 8, ll. 1-24; p. 9, l. 15 - p. 10, l. 25; Figs. 2, 2a, 2b)

expanding at least one term associated with at least one NOT operator into at least one negative term and at least one positive term; and (*see, e.g.,* p. 7, ll. 1-10; p. 9, l. 13 - p. 10, l. 6)

if the service query comprises a term having at least two NOT operators, deleting from the sum of terms a third-order term corresponding to the term having at least two NOT operators; (*see, e.g.,* p. 10, l. 7 - p. 11, l. 11).

evaluating the sum of terms as a plurality of SQL instructions; (*see, e.g.,* p. 6, ll. 23-30; p. 7, ll. 1-10; p. 8, ll. 8-29; Figs. 2, 2a, 2b)

obtaining a plurality of results wherein each term of the sum of terms is associated with one or more results; (*see, e.g.,* p. 6, ll. 14-22; p. 10, ll. 3-25; Figs. 2, 2a, 2b)

generating a first list comprising one or more results associated with the at least one negative term; (*see, e.g.,* p. 10, ll. 3-6; p. 10, ll. 16-22)

generating a second list comprising one or more results associated with the at least one positive term, wherein the first list and the second list are generated by a processor; (*see, e.g.,* p. 10, ll. 3-6; p. 10, ll. 16-22; p. 6, ll. 23-30)

removing or omitting from the second list one or more results associated with the at least one negative term; and (*see, e.g.,* p. 10, ll. 3-6; p. 10, ll. 16-22; Fig. 2b)

providing one or more results from the second list to a user. (*see, e.g.,* p. 1, ll. 15-22)

Grounds of Rejection to be Reviewed on Appeal

Appellant requests the Board to review:

1) the rejection of Claims 1, 9, 14, 22, and 31 under the first paragraph of 35 U.S.C. § 112;

2) the rejection of Claims 1, 9, 14, 22, and 31 under the second paragraph of 35 U.S.C. § 112;

3) the rejection of Claims 1-5, 9-10, 14-18, 22-23, and 27-31 under 35 U.S.C. § 101; and

4) the rejection of Claims 1-5, 9-10, 14-18, 22-23, and 27-31 under 35 U.S.C. § 103(a) as being unpatentable over U.S. Patent No. 6,356,892 B1 issued to Corn, et al. (*"Corn"*), in view of U.S. Patent No. 6,112,198 issued to Lohman, et al. (*"Lohman"*), and in view of U.S. Patent No. 5,412,804 issued to Krishna (*"Krishna"*).

Argument

For at least the following reasons, the Examiner's rejections of Claims 1-5, 9, 10, 14-18, 22, 23 and 27-31 are improper and should be reversed.

I. Claims 1, 9, 14, 22, and 31 satisfy the requirements of 35 U.S.C. § 112, first paragraph.

The Office Action rejects Claims 1, 9, 14, 22, and 31 under Section 112, first paragraph, and alleges that these claims do not comply with the enablement requirement. In particular, the Office Action asserts that "determining a plurality of results...wherein the plurality of results are determined by a processor," as recited in Claim 1, is "not described in the specification in such a way as to enable one skilled in the art to which it pertains, or with which it is most nearly connected, to make and/or use the invention." (Office Action, p. 2). As shown below, this assertion is incorrect.

"The test of enablement is whether one reasonably skilled in the art could make or use the invention from the disclosures in the patent coupled with the information known in the art without undue experimentation." M.P.E.P. § 2164.01 citing *United States v. Teletronics, Inc.*, 857 F.2d 778, 785 (Fed. Cir. 1988). There is no requirement that the specification provide concrete examples or illustrations of claimed steps. In fact, "[c]ompliance with the enablement requirement of 35 U.S.C. § 112, first paragraph, does not turn on whether an example is disclosed." M.P.E.P. § 2164.02. All that is required is that the "information contained in the disclosure of an application must be sufficient to inform those skilled in the relevant art how to both make and use the claimed invention. Detailed procedures for making and using the invention may not be necessary if the description is sufficient to permit those skilled in the art to make and use the invention." M.P.E.P. § 2164.

The Specification provides sufficient information and detail to enable those skilled in the art at the time of invention to make and use the claimed invention. The portion of Claim 1 cited by the Examiner -- "the plurality of results are determined by a processor" -- is supported by the Specification. For example, the Specification describes an "apparatus for processing a directory service query" in order to collect "results" of the query. (*See, e.g.,* p. 6, ll. 14-22). In addition, the Specification provides several detailed examples regarding how to determine results for a directory service query. (*See, e.g.,* p. 7, l. 11-29; p. 10, l. 7 – p. 11,

l. 11).[1] Furthermore, the Specification describes "Relational Database Management Systems" as well as "systems used to provide directory services, such as X.500 service directories and LDAP service directories." (*See, e.g.,* p. 1, ll. 4-8). Accordingly, the Specification is sufficient to inform those skilled in the relevant art how to both make and use the claimed invention. Therefore, the rejection is improper and Appellant respectfully requests the Board to reverse the rejection of Claim 1.

In rejecting Claims 9, 14, 22, and 31 under Section 112, first paragraph, the Office Action employs the same rationale used to reject Claim 1. Accordingly, for reasons analogous to those stated above with respect to Claim 1, Appellant respectfully requests the Board to reverse the rejections of Claims 9, 14, 22, and 31.

II. Claims 1, 9, 14, 22, and 31 satisfy the requirements of 35 U.S.C. § 112, second paragraph.

The Office Action improperly asserts that two elements of Claim 1 are indefinite. As shown below, these two elements are clear and adequately supported by the specification.

A. "a first list...and...a second list"

The Office Action improperly asserts that the following portion of Claim 1 is indefinite:

> ...the determination comprises:
>
> collecting, into a first list, results associated with the one or more negative terms, and
>
> collecting, into a second list, results associated with the one or more positive terms while omitting from the second list any results that are in the first list....

To support the rejection, the Examiner merely quotes the following portion of the Specification:

> It should be noted that a database that supports SQL may not supply a subtraction operator. In such instances a problem in processing the sum of terms as described above may arise. In order to process (evaluate) a subtraction, the method according to the present application: collects all positive terms in a list; collects all negative terms into another list; and then

[1] In citing these portions of the Specification, Appellant does not intend to limit the claims to any particular embodiment. These portions of the Specification are merely cited to illustrate that Claim 1 complies with 35 U.S.C. § 112.

> subtracts the positive term list and the negative term list whilst ignoring duplicates.

(Office Action, p. 3) (quoting Specification, p. 9, l. 27 –p. 10, l. 2). The Office Action does not explain how or why the above portion of the Specification renders Claim 1 indefinite. Indeed, the cited elements of Claim 1 are clear and are sufficiently supported by the Specification. For example, another portion of the Specification, ignored by the Examiner, states:

> An alternative to the subtraction process noted above, is to collect all negative terms in a list, and in the process of collecting all positive terms in another list, only keep the terms that are not in the negative list.

(Specification, p. 10, ll. 3-5).[2] Thus, the Specification describes an alternative to subtraction. At least this portion of the Specification supports "collecting, into a first list, results associated with the one or more negative terms, and collecting, into a second list, results associated with the one or more positive terms while omitting from the second list any results that are in the first list" as recited in Claim 1. Because Claim 1 is definite and adequately supported by the Specification, the rejection should be reversed.

In rejecting Claims 9, 14, 22, and 31 under Section 112, second paragraph, the Office Action employs the same rationale used to reject Claim 1. Accordingly, for reasons analogous to those stated above with respect to Claim 1, Appellant respectfully requests the Board to reverse the rejections of Claims 9, 14, 22, and 31.

B. "the plurality of results are determined by a processor"

The Office Action improperly asserts that another portion of Claim 1 -- "the plurality of results are determined by a processor" -- is indefinite. (Office Action, p. 3). The Office Action offers no explanation or support for this assertion. As shown below, the rejection is improper.

The essential inquiry for determining compliance with 35 U.S.C. § 112, second paragraph, is "whether the claims set out and circumscribe a particular subject matter with a reasonable degree of clarity and particularity." M.P.E.P. § 2173.02. "The requirement to 'distinctly' claim means that the claim must have a meaning discernable to one or ordinary

[2] In citing this portion of the Specification, Applicant does not intend to limit the claims to any particular embodiment. This citation merely shows that the disputed portion of Claim 1 is supported by at least a portion of the Specification.

skill in the art when construed according to correct principles....*Only* when a claim remains insolubly ambiguous without a discernable meaning after all reasonable attempts at construction must a court declare it indefinite." *Metabolite Labs. v. Lab. Corp. of Am. Holdings,* 370 F.3d 1354, 1366, 71 U.S.P.Q.2d 1081, 1089 (Fed. Cir. 2004) (emphasis added). Furthermore, "a claim term that is not used...in the specification is not indefinite if the meaning of the claim term is discernable." *Bancorp Services, L.L.C. v. Hartford Life Ins. Co.*, 359 F.3d 1367, 1372, 69 U.S.P.Q.2d 1996, 1999-2000 (Fed. Cir. 2004).

The cited portion of Claim 1 -- "the plurality of results are determined by a processor" -- is clear. In addition, the claim language is supported by the Specification. The Specification provides several detailed examples regarding how to determine results for a directory service query. (*See, e.g.,* p. 7, l. 11-29; p. 10, l. 7 – p. 11, l. 11). The Specification describes "Relational Database Management Systems" as well as "systems used to provide directory services, such as X.500 service directories and LDAP service directories." (*See, e.g.,* p. 1, ll. 4-8). The Specification discloses an "apparatus for processing a directory service query" to collect "results" of the query. (*See, e.g.,* p. 6, ll. 14-22). Thus, the cited portion of Claim 1 -- "the plurality of results are determined by a processor" -- is supported by the Specification and is clear to one of ordinary skill in the art. Accordingly, the rejection is improper and should be reversed.

In rejecting Claims 9, 14, 22, and 31 under Section 112, second paragraph, the Office Action employs the same rationale used to reject Claim 1. Accordingly, for reasons analogous to those stated above with respect to Claim 1, Appellant respectfully requests the Board to reverse the rejections of Claims 9, 14, 22, and 31.

III. Claims 1-5, 9-10, 14-18, 22-23, and 27-31 satisfy the requirements of 35 U.S.C. § 101.

The Office Action rejects Claims 1-5, 9-10, 14-18, 22-23, and 27-31 under 35 U.S.C. § 101. (Office Action, p. 4). As shown below, this rejection is improper.

A. Claims 1-5, 14-18, and 27-30

The Office Action asserts that Claim 1 is merely directed to functional or nonfunctional descriptive material. (Office Action, p. 4). This assertion is incorrect. Claim

1 is directed to a "method of processing a database service query." Specifically, Claim 1 recites:

> A method of processing a database service query, comprising:
> receiving a service query,
> obtaining a sum of terms associated with the service query by expanding at least one nested term into one or more un-nested terms,
> evaluating the sum of terms as a plurality of SQL instructions, wherein the sum of terms comprises one or more positive terms and one or more negative terms,
> determining a plurality of results associated with the sum of terms, wherein the plurality of results are determined by a processor and the determination comprises:
> collecting, into a first list, results associated with the one or more negative terms, and
> collecting, into a second list, results associated with the one or more positive terms while omitting from the second list any results that are in the first list,
> and
> providing one or more results from the second list to a user.

The method recited in Claim 1 is patentable. In the recent *Comiskey* decision, the Federal Circuit held that "[w]hen an unpatentable mental process is combined with a machine, the combination may produce patentable subject matter." *In re Comiskey*, 499 F.3d 1365, 1379 (Fed. Cir. 2007). In *Comiskey*, the Examiner rejected all of the claims in an application directed to a system for mandatory arbitration. *Id.* at 1368. The Federal Circuit, however, observed that some of the claims (i.e., claims 17 and 46) recited the use of "modules" (e.g., a "registration module for enrolling", "an arbitration module for incorporating arbitration language", etc.). *Id.* at 1379. Accordingly, the Federal Circuit concluded that these "claims, under the broadest reasonable interpretation, could require the use of a computer." *Id.* Based on this conclusion, the Federal Circuit held that claims 17 and 46, in "combining the use of machines with a mental process, claim patentable subject matter." *Id.* at 1380. Therefore, in *Comiskey*, the Federal Circuit reversed the rejection of claims 17 and 46 under 35 U.S.C. § 101. *Id.* at 1381.

This aspect of *Comiskey* is consistent with other Federal Circuit decisions which held that claims combining a mental process with a machine are patentable. *See, e.g., State St. Bank & Trust Co. v. Signature Fin. Group, Inc.*, 149 F.3d 1368, 1371 (Fed. Cir. 1998) (holding patentable a "system that allows an administrator to monitor and record the financial information flow and make all calculations necessary for maintaining a partner fund financial services configuration" where a "computer or equivalent device [wa]s a virtual necessity to perform the task").

Like claims 17 and 46 in *Comiskey*, Claim 1 in the present application recites a machine or the use of a machine as part of the claimed invention. In particular, Claim 1 recites that "the plurality of results are determined by a processor." Therefore, under the Federal Circuit's decision in *Comiskey*, Claim 1 is directed to patent eligible subject matter. Furthermore, the Examiner does not dispute that the method in Claim 1 provides a useful, concrete, and tangible result. Accordingly, Appellant respectfully requests the Board to reverse the rejection of Claim 1.

In rejecting Claims 2-3, 5, 14-16, 18, and 27-30, the Examiner employs the same rationale used to reject Claim 1. Accordingly, for reasons analogous to those stated above with respect to Claim 1, Appellant respectfully requests the Board to reverse the rejection of Claims 2-3, 5, 14-16, 18, and 27-30.

B. Claims 9-10, 22-23, and 31

The Office Action rejects Claims 9-10, 22-23, and 31 under 35 U.S.C. § 101 and asserts that these claims "lack the necessary physical articles or objects to constitute a machine or manufacture." (Office Action, p. 4). This assertion is incorrect. Claim 9 is directed to a "system for processing a directory service query." Specifically, Claim 9 recites:

> A system for processing a directory service query, comprising:
> a database operable to store arbitrary data; and
> a processor that is communicatively coupled to the database and that processes a service query by:
> obtaining a sum of terms by expanding at least one nested term into one or more un-nested terms,
> evaluating the sum of terms as a plurality of SQL instructions, wherein the sum of terms comprises one or more positive terms and one or more negative terms,
> determining a plurality of results associated with the sum of terms, wherein the determination comprises:
> collecting, into a first list, results associated with the one or more negative terms, and
> collecting, into a second list, results associated with the one or more positive terms while omitting from the second list any results that are in the first list,
> and
> providing one or more results from the second list to a user.

Thus, Claim 9 is directed a "system" that comprises "a database" and "a processor that...processes a service query." A "system" comprising a "database" and a "processor" is

clearly a machine or manufacture within the meaning of 35 U.S.C. § 101. The Examiner does not dispute that the system in Claim 9 provides a practical, useful, and concrete result. Accordingly, Claim 9 is directed to patent eligible subject matter. Therefore, Appellant respectfully requests the Board to reverse the rejection of Claim 9.

In rejecting Claims 10, 22, 23, and 31 the Examiner employs the same rationale used to reject Claim 1. Accordingly, for reasons analogous to those stated above with respect to Claim 1, Appellant respectfully requests the Board to reverse the rejection of Claims 10, 22, 23, and 31.

IV. Claims 1-5, 9-10, 14-18, 22-23, and 27-31 are allowable over the cited references.

The Office Action rejects Claims 1-5, 9-10, 14-18, 22-23, and 27-31 under 35 U.S.C. § 103(a) as being unpatentable over U.S. Patent No. 6,356,892 B1 issued to Corn, et al. (*"Corn"*), in view of U.S. Patent No. 6,112,198 issued to Lohman, et al. (*"Lohman"*), and in view of U.S. Patent No. 5,412,804 issued to Krishna (*"Krishna"*). As shown below, the rejection is improper.

A. Claims 1, 3-5, 9-10, 14, 16-18, 22-23, 27-28, and 30

The Office Action fails to support the rejection of Claim 1 for several reasons. First, the cited references fail to teach, suggest, or disclose that the "determination comprises...collecting, into a first list, results associated with the one or more negative terms, and collecting, into a second list, results associated with the one or more positive terms" as recited in Claim 1. Second, the cited references fail to teach, suggest, or disclose "omitting from the second list any results that are in the first list" as recited in Claim 1. Third, the *Corn-Lohman* combination is improper because the proposed combination would render *Corn* unsatisfactory for its intended purpose.

First, the cited references fail to teach, suggest, or disclose that the "determination comprises...collecting, into a first list, results associated with the one or more negative terms, and collecting, into a second list, results associated with the one or more positive terms" as recited in Claim 1. In the Office Action, the Examiner relies on *Corn* for this aspect of Claim 1. (Office Action, pp. 6-7). *Corn* teaches a method for re-writing LDAP queries as SQL

queries. (*Corn*; abstract). The cited portion of *Corn* describes merging sets of EIDs into an SQL query. (Col. 7, ll. 39-58). In particular, the cited portion of *Corn* states:

> As described above, according to the inventive method, for each LDAP filter element or sub-expression, there is a set of entries (EIDs) that will satisfy the element. Thus, each element generally maps to a set of EIDs. The EID sets are then merged together, preferably into a single SQL query, using a set of combination rules. Thus, if a pair of LDAP filter elements are subject to an LDAP logical OR operator, the corresponding EID sets are merged using an SQL UNION logical operator. If a pair of LDAP filter elements are subject to an LDAP logical AND operator, the corresponding EID sets are merged using an SQL INTERSECT logical operator. If a pair of LDAP filter elements are subject to an LDAP logical NOT operator, the corresponding EID sets are merged using an SQL NOT IN logical operator. As will also be seen, these combination rules are applied recursively such that all LDAP elements associated with a particular logical operator are processed into the SQL query. This recursive processing facilitates handling of even complicated LDAP queries having numerous layers of logical depth.

(Col. 7, ll. 39-58). Thus, *Corn* describes merging sets of EIDs "using a set of combination rules." *Id.* This portion of *Corn*, however, does not teach, suggest, or disclose "results associated with the one or more negative terms" or "results associated with the one or more positive terms" as recited in Claim 1. In addition, merely merging sets of EIDs, as described in *Corn*, does not teach, suggest, or disclose "collecting, *into a first list*, results associated with the one or more negative terms, and collecting, *into a second list*, results associated with the one or more positive terms" as recited in Claim 1. (Emphases added.) Because the cited references fail to teach, suggest, or disclose this aspect of Claim 1, the cited references fail to support the rejection.

Second, the cited references fail to teach, suggest, or disclose "omitting from the second list any results that are in the first list" as recited in Claim 1. In the Office Action, the Examiner relies on *Lohman* for this aspect of Claim 1. (Office Action, p. 7). *Lohman* describes a method for parallel processing of subtasks associated with a query. (Abstract; col. 1, ll. 42-59). The cited portion of *Lohman* describes duplicate elimination. (Col. 5, ll. 30-33). Specifically, the cited portion states:

> Duplicate elimination (distinct) can be thought of as a special case of aggregation (with no aggregating functions); therefore, all of the parallelization decisions just described for aggregation apply to duplicate elimination as well.

(Col. 5, ll. 30-34). Thus, the cited portion of *Lohman* includes a cursory reference to "duplicate elimination." Neither the cited portions of *Corn* or *Lohman* teach, suggest, or

disclose the "first list" or the "second list" recited in Claim 1. Therefore, the cursory reference to "duplicate elimination" in *Lohman* does not teach, suggest, or disclose "omitting from the second list any results that are in the first list" as recited in Claim 1. Because the cited references fail to teach, suggest, or disclose this aspect of Claim 1, the cited references fail to support the rejection.

Third, the *Corn-Lohman* combination is improper because the proposed combination would render *Corn* unsatisfactory for its intended purpose. If a "proposed modification would render the prior invention being modified unsatisfactory for its intended purpose, then there is no suggestion or motivation to make the proposed modification." M.P.E.P. § 2143.01. A "primary object" of *Corn* is to search a relational database to retrieve "target entries that exactly match given search criteria." (*Corn*; col. 2., ll. 32-38). *Corn* describes a method for mapping an LDAP search query into an SQL query. (*Corn*; col. 2, ll. 51-54). To formulate the SQL query, the method in *Corn* must generate "unique entry identifier (EID) sets." (*Corn*; col. 3, ll. 1-20). "Each LDAP entry is assigned a unique identifier (EID)" and the EIDs are stored in an "Entry Table" and an "Attribute Table." (*Corn*; col. 5, ll. 59-65; col. 6, ll. 16-22). The "Entry Table" stores each EID with entry data, and the "Attribute Table" stores each EID with attribute values. (*Corn*; col. 5, ll. 58-67; col. 6, ll. 16-22). *Corn* formulates SQL queries by incorporating terms associated with EIDs. In particular, *Corn* requires that the EID sets be "merged together, preferably into a single SQL query." (*Corn*; col. 7, ll. 42-44). For example, *Corn* presents the following SQL query:

> SELECT entry.EntryData, FROM LDAP_ENTRY as entry WHERE entry.**EID** in (SELECT distinct LDAP_ENTRY.**EID** FROM LDAP_ENTRY.ldap_desc WHERE (LDAP_ENTRY.**EID**=ldap_desc.DEID AND ldap_desc.AEID=<id>) AND LDAP_ENTRY.**EID** NOT IN ((SELECT **EID** FROM f1 where f1=` v1`))).

(*Corn*; col. 11, ll. 10-20) (emphases added). Notably, this SQL query comprises numerous terms associated with EIDs. Therefore, to retrieve data using this query, *Corn* must search the EIDs associated with LDAP entries.

In contrast to *Corn*, the method in *Lohman* does not generate or store EIDs for each LDAP entry. Rather, *Lohman* describes a method for separating a query into "subtasks" and dividing a database into multiple "partitions." (*Lohman*; col. 2, ll. 53-60). *Lohman* then applies each subtask to an individual partition of the database. (*Lohman*; col. 2, ll. 53-60). Combining *Corn* with *Lohman* would result in applying queries associated with EIDs to a

database without EIDs. A query or subquery for particular EIDs, as described in *Corn*, will simply not return results from a database that does not comprise any EIDs, as described in *Lohman*. Thus, the proposed combination would render *Corn* inoperable and therefore unsatisfactory for its intended purpose of retrieving "target entries that exactly match given search criteria." (*Corn*; col. 2., ll. 32-38). Because the proposed combination would render *Corn* unsatisfactory for its intended purpose, the combination is improper and should be withdrawn. For at least the foregoing reasons, Appellant respectfully requests the Board to reverse the rejection of Claim 1.

In rejecting Claims 9, 14, and 22, the Examiner employs the same rationale used to reject Claim 1. Accordingly, for reasons analogous to those stated above with respect to Claim 1, Appellant respectfully requests the Examiner to reverse the rejections of Claims 9, 14, and 22.

Claims 3-5, 10, 16-18, 23, 27-28, and 30 depend from independent claims shown above to be allowable. In addition, these claims recite further elements that are not taught, suggested, or disclosed by the cited references. Accordingly, Appellant respectfully requests the Board to reverse the rejections of Claims 3-5, 10, 16-18, 23, 27-28, and 30.

B. Claim 31

The cited references fail to teach, suggest, or disclose each element of Claim 31. For example, the cited references fail to teach, suggest, or disclose "if the service query comprises a term having at least two NOT operators, deleting from the sum of terms a third-order term corresponding to the term having at least two NOT operators" as recited in Claim 31. In the Office Action, the Examiner relies on *Corn* for this aspect of Claim 31. (Office Action, p. 17). In particular, the Examiner cites a flowchart (Figure 6B of *Corn*) as well as the description in *Corn* that pertains to the flowchart. *Corn* explains that the flowchart illustrates a SQL generation algorithm. (Col. 7, ll. 62-64). The cited portion of *Corn* states:

> If the LDAP filter element includes neither AND nor OR, the routine continues at step 92 to determine whether the NOT logical operator is present. If so, the routine continues at step 94 to add the NOT IN logical operator to the SQL expression being generated. The routine then continues at step 96 to enter the recursive call so that all associated subexpressions may be parsed through the algorithm in the manner previously described. Thus, at step 98, a test is performed to determine whether all subexpressions associated with the NOT operator have been processed. If so, the routine returns at step 100;

> otherwise, the routine loops back to step 96 and processes the next subexpression.

(Col. 8, ll. 40-50). Thus, the cited portion of *Corn* describes parsing subexpressions and, in some cases, adding the NOT IN logical operator to a SQL expression. This reference to parsing subexpressions, however, does not teach, suggest, or disclose "a term having at least two NOT operators" or "a third-order term corresponding to the term having at least two NOT operators" as recited in Claim 31. Furthermore, there is nothing in the cited portion of *Lohman* that teaches, suggests, or discloses "if the service query comprises a term having at least two NOT operators, *deleting from the sum of terms a third-order term* corresponding to the term having at least two NOT operators" as recited in Claim 31. (Emphasis added.) Because the cited references fail to teach, suggest, or disclose this aspect of Claim 31, the cited references fail to support the rejection. Accordingly, Appellant respectfully requests the Examiner to reverse the rejection of Claim 31.

C. Claims 2 and 15

The cited references fail to teach, suggest, or disclose each element of Claim 2. For example, the cited references fail to teach, suggest, or disclose "expanding each term to remove NOT operators" as recited, in part, in Claim 2. The Examiner relies on *Corn* for this aspect of Claim 2. (Office Action, p. 7). *Corn* teaches a method for re-writing LDAP queries as SQL subqueries. (*Corn*; Abstract). The cited portion of *Corn* describes re-writing an LDAP logical operator NOT to an SQL logical operator NOT IN. (*Corn*; col. 8, ll. 40-51). Specifically, the cited portion of *Corn* states:

> If a pair of LDAP filter elements are subject to an LDAP logical NOT operator, the corresponding EID sets are merged using an SQL NOT IN logical operator.

(*Corn*; col. 7, ll. 50-52). Thus, *Corn* teaches re-writing a query term comprising a NOT operator as an SQL term comprising a NOT IN logical operator. Simply re-writing a query term to comprise a NOT IN operator does not teach, suggest, or disclose removing "NOT operators" as recited in Claim 2. Thus, the cited portion of *Corn* does not support the rejection of Claim 2. Accordingly, Appellant respectfully requests the Board to reverse the rejection of Claim 2.

In rejecting Claim 15, the Examiner employs the same rationale used to reject Claim 2. Accordingly, for reasons analogous to those stated above with respect to Claim 2, Appellant respectfully requests the Board to reverse the rejection of Claim 15.

D. Claim 29

The cited references fail to teach, suggest, or disclose each element of Claim 29. For example, the cited references fail to teach, suggest, or disclose "if the service query comprises a term having at least two NOT operators, deleting or disregarding from the sum of terms a third-order term corresponding to the term having at least two NOT operators" as recited in Claim 29. In the Office Action, the Examiner cites *Corn* with respect to this aspect of Claim 29. (Office Action, p. 16). The cited portion of *Corn* states:

> If the LDAP logical operator is NOT, the invention preferably excludes entries by negating the IN operation before the subquery. Thus, the combination rules includes, for example, mapping the LDAP logical OR operation to an SQL UNION, mapping the LDAP logical operation AND to SQL INTERCEPT, and mapping the LDAP logical operation NOT to SQL NOT IN.

(*Corn*; col. 3, ll. 14-20). This portion of *Corn* merely teaches mapping the LDAP operator NOT to the SQL operator NOT IN. The cited portion of *Corn* makes no mention of "a third-order term" or of "a term having at least two NOT operators" as recited in Claim 29. In addition, the cited portion of *Corn* clearly fails to teach, suggest, or disclose "deleting or disregarding from the sum of terms a third-order term corresponding to the term having at least two NOT operators" as recited in Claim 29. Because the cited references fail to teach, suggest, or disclose the foregoing aspects of Claim 29, Appellant respectfully requests the Board to reverse the rejection of Claim 29.

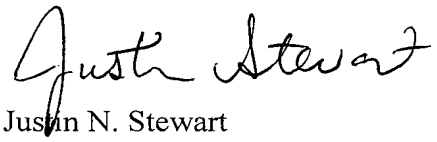
Conclusion

Appellant respectfully requests the Board of Patent Appeals and Interferences to reverse the rejection of Claims 1-5, 9, 10, 14-18, 22, 23 and 27-31 and instruct the Examiner to issue a notice of allowance as to all pending claims.

The Commissioner is hereby authorized to charge the large entity fee of $510.00 under 37 C.F.R. § 41.20(b)(2) for filing this Appeal Brief and the $120.00 one-month extension fee to Deposit Account No. 02-0384 of Baker Botts L.L.P. Although no other fees are believed to be due at this time, the Commissioner is hereby authorized to charge any additional fees and/or credit any overpayments to Deposit Account No. 02-0384 of Baker Botts L.L.P.

Respectfully submitted,

BAKER BOTTS L.L.P.
Attorneys for Appellant



Justin N. Stewart
Reg. No. 56,449
(214) 953-6755

Date: September 30, 2008

CORRESPONDENCE ADDRESS:

Customer No. **05073**

Appendix A: Claims on Appeal

1. **(Previously Presented)** A method of processing a database service query, comprising:

receiving a service query,

obtaining a sum of terms associated with the service query by expanding at least one nested term into one or more un-nested terms,

evaluating the sum of terms as a plurality of SQL instructions, wherein the sum of terms comprises one or more positive terms and one or more negative terms,

determining a plurality of results associated with the sum of terms, wherein the plurality of results are determined by a processor and the determination comprises:

collecting, into a first list, results associated with the one or more negative terms, and

collecting, into a second list, results associated with the one or more positive terms while omitting from the second list any results that are in the first list,

and

providing one or more results from the second list to a user.

2. **(Original)** The method as claimed in claim 1, further comprising expanding each term to remove NOT operators.

3. **(Original)** The method as claimed in claim 2, wherein the sum of terms are expanded using Boolean logic.

4. **(Original)** The method as claimed in claim 1, in which the service query is an X.500 or LDAP service query.

5. **(Original)** The method as claimed in claim 1, in which the service query is a search service query.

6. **(Withdrawn)** A method of processing a database service query, comprising:

determining a SQL instruction representative of a function;

listing the results of a subtracted SQL instruction in a first list, listing the results of a non-subtracted SQL instruction in a second list; and

not listing results which are duplicates of previously listed subtracted or non-subtracted results.

7. **(Withdrawn)** The method as claimed in claim 6, in which the service query is an X.500 or LDAP query.

8. **(Withdrawn)** The method as claimed in claim 6, in which the service query is a search service query.

9. **(Previously Presented)** A system for processing a directory service query, comprising:

a database operable to store arbitrary data; and

a processor that is communicatively coupled to the database and that processes a service query by:

obtaining a sum of terms by expanding at least one nested term into one or more un-nested terms,

evaluating the sum of terms as a plurality of SQL instructions, wherein the sum of terms comprises one or more positive terms and one or more negative terms,

determining a plurality of results associated with the sum of terms, wherein the determination comprises:

collecting, into a first list, results associated with the one or more negative terms, and

collecting, into a second list, results associated with the one or more positive terms while omitting from the second list any results that are in the first list,

and

providing one or more results from the second list to a user.

10. **(Previously Presented)** The system as claimed in claim 9, further comprising means to perform X.500 or LDAP services.

11. **(Withdrawn)** A directory service arrangement comprising:

a database using a plurality of tables, each table having a plurality of rows and columns, and storing arbitrary data, and

means for processing a service query by determining a SQL instruction representative of a function, listing the results of a subtracted SQL instruction in a first list, listing the results of a non-subtracted SQL instruction in a second list, and not listing results which are duplicates of previously listed subtracted or non-subtracted results.

12. **(Withdrawn)** The directory service arrangement as claimed in claim 11, further comprising means to perform X.500 or LDAP services.

13. **(Withdrawn)** A method for processing a database service query, comprising:

translating a service query to an expression;

simplifying the expression to a number of smaller expressions, each smaller expression being capable of being flattened;

flattening each smaller expression; and

executing each flattened expression.

14. **(Previously Presented)** A method of processing a directory service query, comprising:

receiving a directory service query,

obtaining a sum of terms by expanding at least one nested term into one or more un-nested terms,

mapping the sum of terms to a plurality of SQL instructions, wherein the sum of terms comprises one or more positive terms and one or more negative terms,

determining a plurality of results associated with the sum of terms, wherein the plurality of results are determined by a processor and the determination comprises:

collecting, into a first list, results associated with the one or more negative terms, and

collecting, into a second list, results associated with the one or more positive terms while omitting from the second list any results that are in the first list,

and

providing one or more results from the second list to a user.

15. **(Previously Presented)** The method as claimed in claim 14, further comprising expanding each term to remove NOT operators.

16. **(Original)** The method as claimed in claim 15, wherein the sum of terms are expanded using Boolean logic.

17. **(Previously Presented)** The method as claimed in claim 14, in which the directory service query is an X.500 or LDAP service query.

18. **(Previously Presented)** The method as claimed in claim 14, in which the directory service query is a search service query.

19. **(Withdrawn)** A method of processing a directory service query, comprising:

determining a SQL instruction representative of the directory service query;

listing the results of a subtracted SQL instruction in a first list, listing the results of a non-subtracted SQL instruction in a second list; and

not listing results which are duplicates of previously listed subtracted or non-subtracted results.

20. **(Withdrawn)** The method as claimed in claim 19, in which the service query is an X.500 or LDAP query.

21. **(Withdrawn)** The method as claimed in claim 19, in which the service query is a search service query.

22. **(Previously Presented)** A system for processing a directory service query, comprising:

a database that is operable to store arbitrary data; and

a processor that is communicatively coupled to the database and that processes a directory service query by:

obtaining a sum of terms by expanding at least one nested term into one or more un-nested terms,

mapping the sum of terms to a plurality of SQL instructions, wherein the sum of terms comprises one or more positive terms and one or more negative terms,

determining a plurality of results associated with the sum of terms, wherein the determination comprises:

collecting, into a first list, results associated with the one or more negative terms, and

collecting, into a second list, results associated with the one or more positive terms while omitting from the second list any results that are in the first list,

and

providing the determined plurality of results to a user.

23. **(Previously Presented)** The system as claimed in claim 22, further comprising means to perform X.500 or LDAP services.

24. **(Withdrawn)** A directory service arrangement comprising:

a database using a plurality of tables, each table having a plurality of rows and columns, and storing arbitrary data, and

means for processing a directory service query by determining a SQL instruction representative of the directory service query, listing the results of a subtracted SQL instruction in a first list, listing the results of a non-subtracted SQL instruction in a second list, and not listing results which are duplicates of previously listed subtracted or non-subtracted results.

25. **(Withdrawn)** The directory service arrangement as claimed in claim 24, further comprising means to perform X.500 or LDAP services.

26. **(Withdrawn)** A method for processing a directory service query, comprising:

translating a directory service query to an expression;

simplifying the expression to a number of smaller expressions, each smaller expression being capable of being flattened;

flattening each smaller expression; and

executing each flattened expression.

27. **(Previously Presented)** The method of claim 1 wherein:

evaluating the sum of terms comprises converting the sum of terms to a plurality of SQL instructions comprising at least one negative term;

and further comprising:

subtracting at least one result associated with the at least one negative term.

28. **(Previously Presented)** The method of claim 1, wherein obtaining a sum of terms comprises:

identifying at least one term associated with at least one NOT operator; and

expanding the at least one term associated with the at least one NOT operator into at least one negative term.

29. **(Previously Presented)** The method of claim 1, wherein:

if the service query comprises a term having at least two NOT operators, deleting or disregarding from the sum of terms a third-order term corresponding to the term having at least two NOT operators.

30. **(Previously Presented)** The method of claim 1, wherein obtaining a sum of terms comprises:

identifying at least one term associated with at least one NOT operator; and

expanding the at least one term associated with at least one NOT operator into at least one negative term and at least one positive term.

31. **(Previously Presented)** A method of processing a database service query, comprising:

receiving a service query;

obtaining a sum of terms associated with the service query by:

expanding at least one nested term into one or more un-nested terms;

expanding at least one term associated with at least one NOT operator into at least one negative term and at least one positive term; and

if the service query comprises a term having at least two NOT operators, deleting from the sum of terms a third-order term corresponding to the term having at least two NOT operators;

evaluating the sum of terms as a plurality of SQL instructions;

obtaining a plurality of results wherein each term of the sum of terms is associated with one or more results;

generating a first list comprising one or more results associated with the at least one negative term;

generating a second list comprising one or more results associated with the at least one positive term, wherein the first list and the second list are generated by a processor;

removing or omitting from the second list one or more results associated with the at least one negative term; and

providing one or more results from the second list to a user.

Appendix B: Evidence

U.S. Patent No. 6,356,892 issued to Corn, et al.

U.S. Patent No. 6,112,198 issued to Lohman, et al.

U.S. Patent No. 5,412,804 issued to Krishna, et al.

Other than the above references, no evidence was submitted pursuant to 37 C.F.R. §§ 1.130, 1.131, or 1.132, and no other evidence was entered by the Examiner and relied upon by Appellant in the Appeal.

US006112198A

United States Patent [19]

Lohman et al.

[11] Patent Number: 6,112,198

[45] Date of Patent: Aug. 29, 2000

[54] **OPTIMIZATION OF DATA REPARTITIONING DURING PARALLEL QUERY OPTIMIZATION**

[75] Inventors: **Guy Maring Lohman; Mir Hamid Pirahesh; Eugene Jon Shekita; David E. Simmen**, all of San Jose; **Monica Sachiye Urata**, Saratoga, all of Calif.

[73] Assignee: **International Business Machines Corporation**, Armonk, N.Y.

[21] Appl. No.: **09/106,473**

[22] Filed: **Jun. 29, 1998**

Related U.S. Application Data

[60] Provisional application No. 60/051,259, Jun. 30, 1997.

[51] **Int. Cl.**[7] **G06F 17/30**

[52] **U.S. Cl.** **707/3**; 707/3

[58] **Field of Search** 707/2, 3, 4

[56] **References Cited**

U.S. PATENT DOCUMENTS

5,745,746	4/1998	Jhingran et al.	707/2
5,828,409	3/1999	Baru et al.	707/2
5,960,427	9/1999	Goel et al.	707/4
5,987,453	11/1999	Krishna et al.	707/4
6,009,265	12/1999	Huang et al.	707/3

OTHER PUBLICATIONS

Baru, C.K., "DB2 Parallel Edition," *IBM Systems Journal*, vol. 34 No. 2, pp. 292–322, 1995.

Lohman, G., "Grammar-like Functional Rules for Representing Query Optimization Alternatives," In *Proceedings of the 1988 ACM SIGMOD Intn'l Conf. on Management of Data*, pp. 18–27, 1988.

Simmen, D., Fundamental Techniques for Order Optimization, In *Proceedings of the 1996 ACM SIGMOD Intn'l Conf. on Management of Data*, pp. 57–67, 1996.

ISO and ANSI SQL3 Working Draft—Feb. 5, 1993, Digital Equipment corporation, Maynard, Massachusetts.

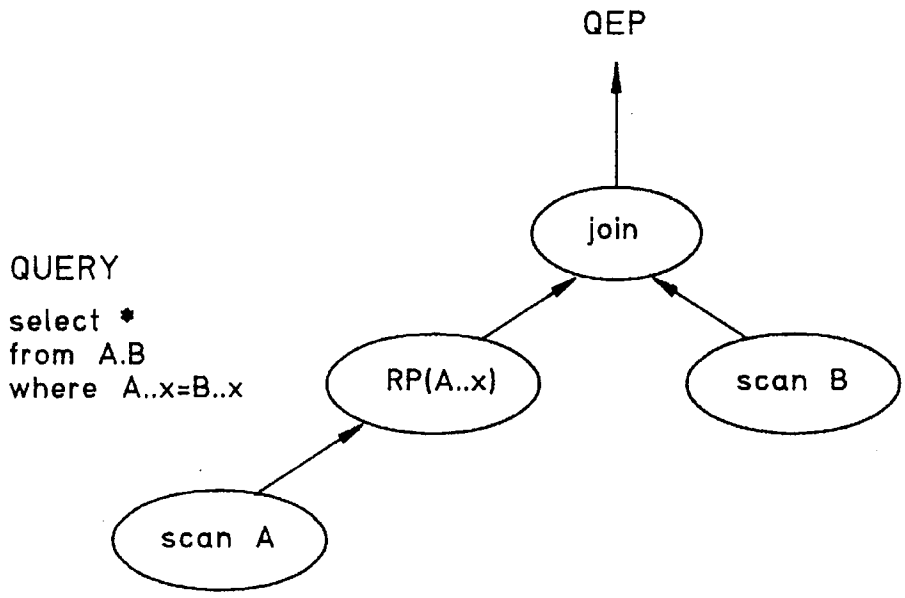
Primary Examiner—Paul V. Kulik

Attorney, Agent, or Firm—Gray Cary Ware Freidenrich

[57] **ABSTRACT**

Query evaluation is optimized using parallel optimization techniques to make repartitioning more efficient. Efficiency is improved by recognizing the possible partitioning requirements for achieving parallelism for a query operation, and by recognizing when the partitioning property of data satisfies the partitioning requirements of a query operation. A data base management system in accordance with the invention uses parallel query processing techniques to optimize data repartitioning, or to avoid it altogether.

44 Claims, 12 Drawing Sheets



Example QEP for a Directed Join

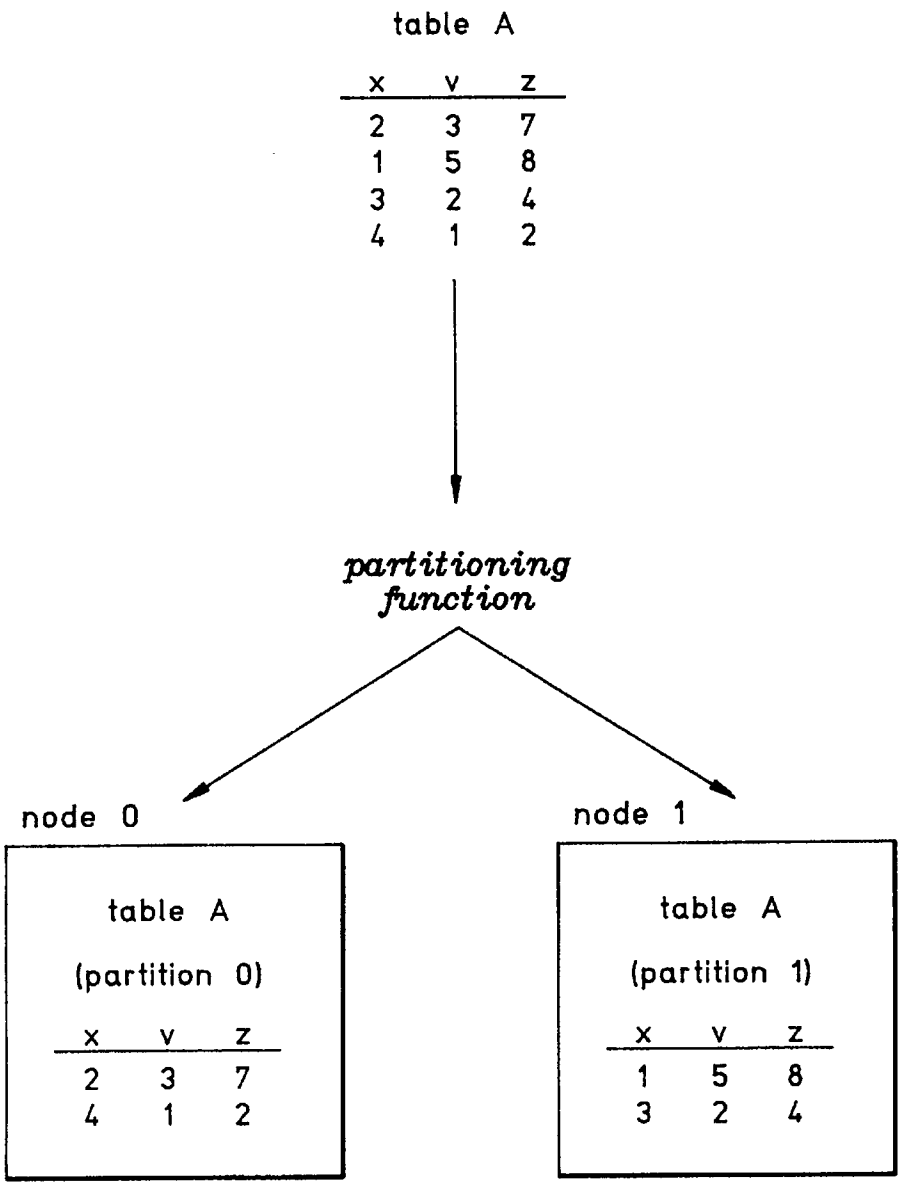


Figure 1: Example of Horizontal Partitioning

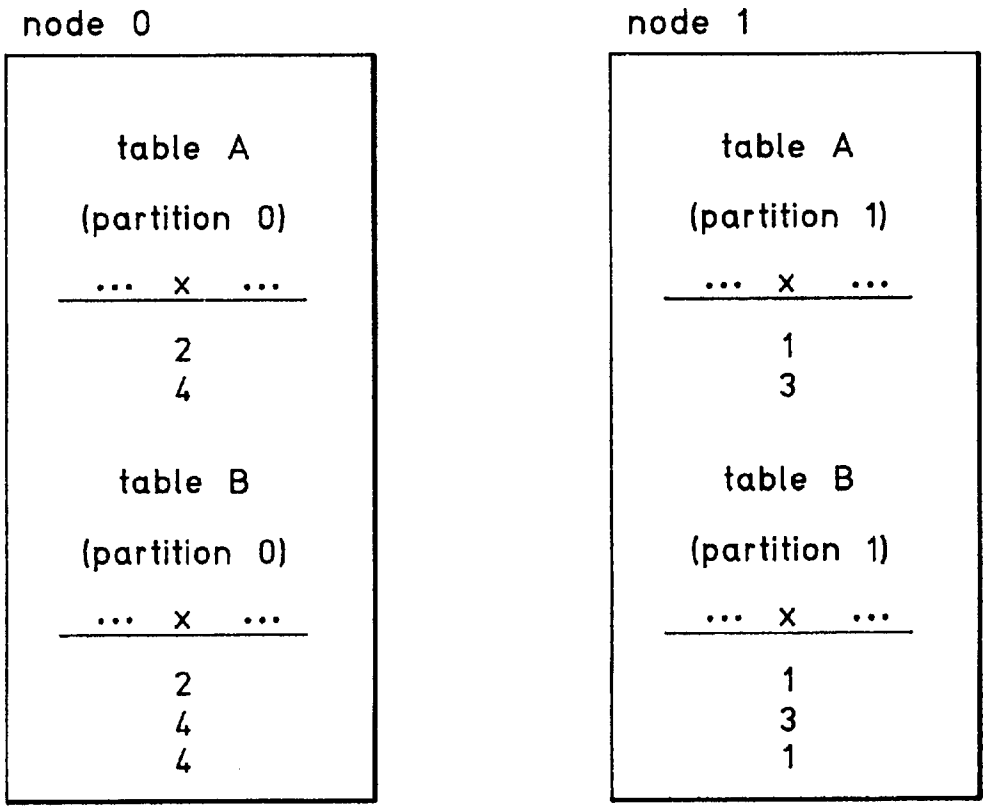


Figure 2: Example of Partitioned Join with no Data Movement

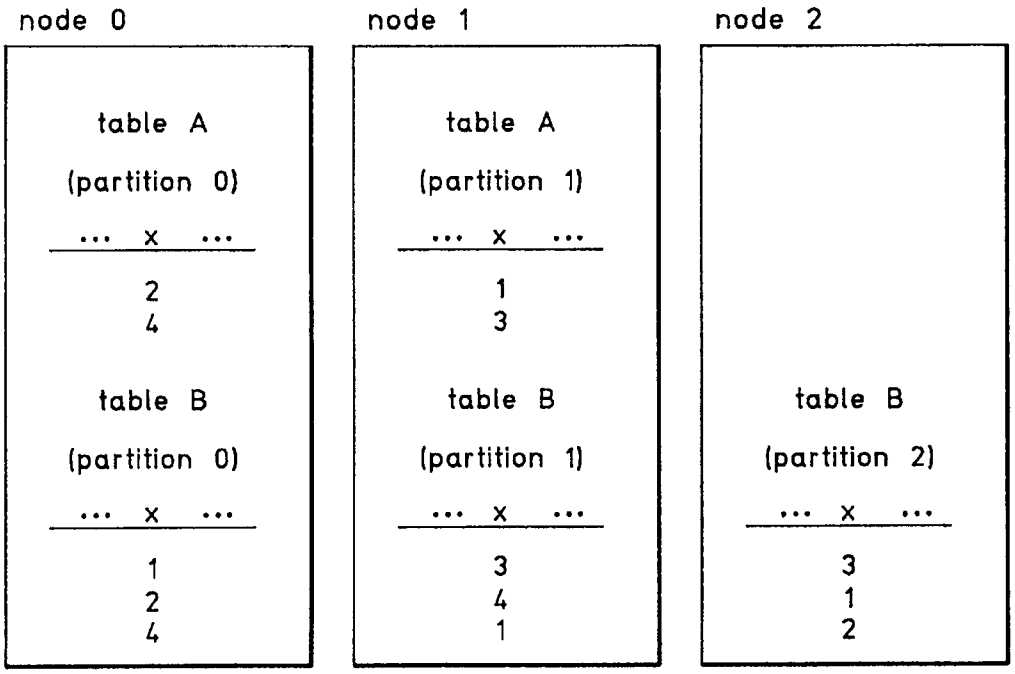


Figure 3: Example of Partitioned Join that Requires Data Movement

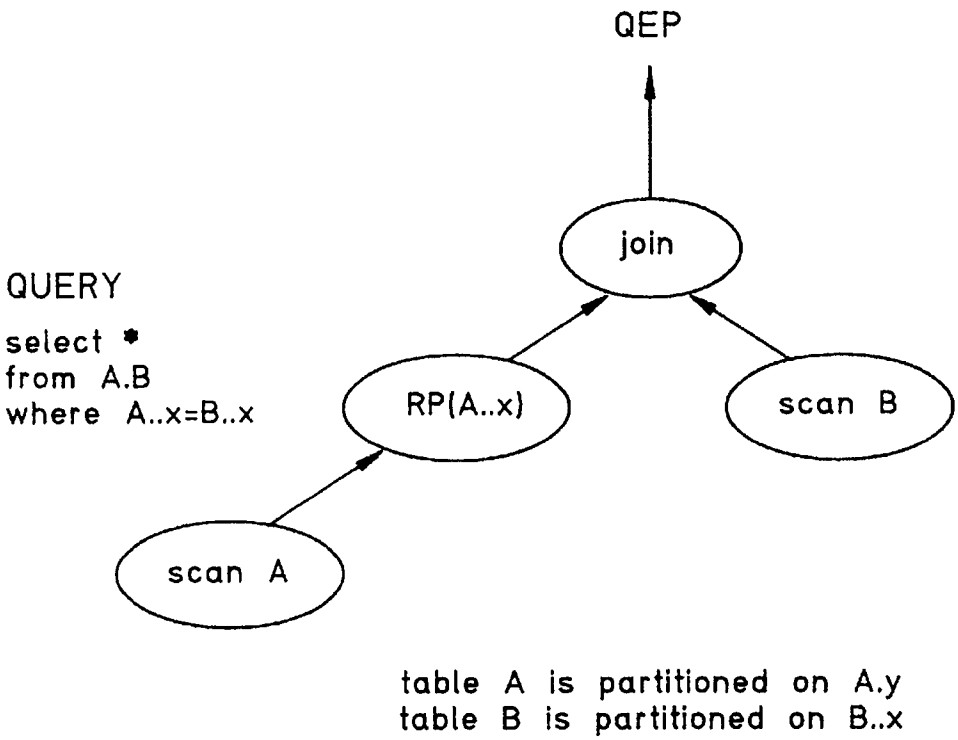


Figure 4: Example QEP for a Directed Join

Figure 5: Column Homogenization Process

```
homogenize
 input:
     applied predicates PREDS,
     target tables TABLES,
     column to homogenize C
 output:
     result of homogenizing the column R

1)   set result R to NULL
2)   compute EQ the equivalence class for C using PREDS
3)   for (each column c_i of EQ while R is NULL)
4)      if (c_i belongs to a table in TABLES) then
5)              set result R to c_i
6)      endif
7)   endfor
8)   return R
```

Figure 5

Figure 6: Mapping a Target Partitioning to Source

```
build-dir-req

input:
        predicates PREDS,
        source tables TABLES,
        target partitioning requirement TP
output:
      source partitioning requirement SP

1)       set b_i to some non NULL value
2)       for (each partitioning column c_i of TP while b_i is not~NULL)
3)             b_i~=~{\bf homogenize~(~PREDS,~TABLES,~c_i)
4)             set the corresponding partitioning column of SP to b_i
5)       endfor
6)       if (b_i~is~not~NULL) then
7)       set the nodegroup identifier of SP to the nodegroup identifier of TP
8)          set the function identifier of SP to the function identifier of TP
9)          return SP
10)      else
11)         return NULL
12)      endif
13)      return SP
```

Figure 6

Figure 7: Using the build–dir–req Process to Build a Join Requirement

```
build-dir-req

input:
        join predicates PREDS,
        target QEP TQEP,
        source tables TABLES
output:
      source partitioning requirement SP

1)      build target part req, TP, by casting part prop of TQEP
2)      SP = build-dir-req (PREDS, TABLES, TP)
3)      if (SP is NULL) then
4)            set the nodegroup i.d. of SP to the nodegroup i.d. of TP
5)            set the function i.d. of SP to "broadcast"
6)      endif
7)      return SP
```

Figure 7

Figure 8: Using build–dir–req Process to Build a Local or Directed Subquery Requirement

```
build-dir-req

input:
      correlation predicates CPREDS,
      subquery predicate SPRED,
      target QEP (applying subquery predicate) TQEP,
      tables of subquery producer TABLES,

output:
      source partitioning requirement SP

1)        build target partitioning requirement, TP, by casting partition property of TQEP
2)        set PREDS to CPREDS
3)        if (SPRED is of form: COL1 OP COL2 where OP
4)              is either = ANY, <> ALL, NOT IN, or IN) then
5)              build predicate COL1 = COL2 and add it to PREDS
6)        endif
7)        SP = build-dir-req (PREDS, TABLES, TP)
8)        if (SP is NULL) then
9)              set the nodegroup i.d. of SP to the nodegroup i.d. of TP
10)             set the function i.d. of SP to "broadcast"
11)       endif
12)       return SP
```

Figure 8

Figure 9: Improved build-dir-req Process

```
build-dir-req
input:
      applied predicates PREDS,
      source tables TABLES,
      target partitioning requirement TP
output:
      source partitioning requirement SP
1)    set b_i to some non NULL value
2)    set all-correlations to true
3)    using PREDS, determine CORR, the set of correlated columns referenced by TABLES
4)    for (each partitioning column c_i of TP while b_i is not~NULL)
5)       if (c_i is bound to a constant b_i) then
6)             set the corresponding partitioning column of SP to b_i
7)       else
8)             b_i = homogenize (PREDS, TABLES, c_i)
9)             if (b_i is not NULL) then
10)                 set the corresponding partitioning column of SP to b_i
11)                 set all-correlations to false
12)            else (if (c_i is in the set CORR) then
13)                 set b_i to c_i
14)                 set the corresponding partitioning column of SP to b_i
15)            endif
16)      endif
17)   endfor
18)   if (b_i~is~not~NULL and all-correlations is FALSE) then
19)   set the nodegroup identifier of SP to the nodegroup identifier of TP
20)      set the function identifier of SP to the function identifier of TP
21)      return SP
22)   else
23)      return NULL
24)   endif
```

Figure 9

Figure 10: Improved build-dir-req Process

```
listener-rp-pred

input:
      tuple to send to caller T
      caller's node number N,
      target partitioning requirement TP
output:
      true if the tuple should be sent to the caller and false otherwise R

1)          set RN to result of applying the partitioning function ot TP to
            columns of T
2)          if (RN is the same as N) then
3)                set R to TRUE
4)          else
5)                set R to FALSE
6)          endif
7)          return R
```

Figure 10

Figure 11: local–agg–test Process

```
local-agg-test

input:
      applied predicates PREDS,
      grouping columns GC
      input QEP for aggregation TQEP
output:
      result of the test R

1)    set result R to FALSE
2)    set PP to the partitioning property of TQEP
3)    if (PP is located on a single node nodegroup)
4)          set result R to TRUE
5)    else
6)          compute the equivalence classes EQ using PREDS
7)          set result R to TRUE
8)          for (each column k_i of the partitioning key of PP while R is TRUE)
9)                set R to FALSE
10)               for (each column c_i of GC while R is FALSE)
11)                  if (c_i and k_i are in the same equivalence class in EQ)
12)                       set result R to TRUE
13)                  endif
14)               endfor
15)         endfor
16)   endif
17)   return R
```

Figure 11

Figure 12: Improved local-agg-test Process

```
local-agg-test
input:
        applied predicates PREDS,
        grouping columns GC
        input QEP for aggregation TQEP
output:
    result of the test R
1)  set result R to FALSE
2)  set PP to the partitioning property of TQEP
3)  if (PP is located on a single node nodegroup)
4)      set result R to TRUE
5)  else
6)      compute the equivalence classes EQ using PREDS
7)      set result R to TRUE
8)      for (each column k_i of the partitioning key of PP while R is TRUE)
9)          if (k_i is not bound to a constant or correlation value)
10)             set R to FALSE
11)             for (each column c_i of GC while R is FALSE)
12)                 if (c_i and k_i are in the same equivalence
                    class EQ)
13)                     set result R to TRUE
14)                 endif
15)             endfor
16)         endif
17)     endfor
18) endif
19) return R
```

Figure 12

OPTIMIZATION OF DATA REPARTITIONING DURING PARALLEL QUERY OPTIMIZATION

This Application claims the benefit of U.S. Provisional Application Ser. No. 60/051,259, filed on Jun. 30, 1997.

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention relates generally to database management systems and, more particularly, to efficient evaluation of queries processed in relational database management systems.

2. Description of the Related Art

A data base management system (DBMS) often uses parallel query execution to deal with the performance demands imposed by applications executing complex queries against massive databases. Such parallelism is frequently achieved by partitioning a database among processors. The queries are broken into subtasks based upon the partitioning of the database, so that different subtasks are assigned to different data partitions. The subtasks are executed by the processor managing the partition and the results of these subtasks are merged for delivery to an end user. Optimization choices regarding how queries are broken into subtasks are driven by how the data is partitioned. That is, the partitioning property of the data determines how the queries are divided into subtasks. Often data has to be repartitioned dynamically to satisfy the partitioning requirements of a given query operation. Repartitioning is an expensive operation and should be optimized or avoided altogether.

From the discussion above, it should be apparent that there is a need for a database management system that evaluates complex query statements with reduced requirements for repartitioning of data and more efficient partitioning of the data. The present invention fulfills this need.

SUMMARY OF THE INVENTION

The present invention optimizes query evaluation by using parallel optimization techniques to optimize repartitioning by recognizing: (1) the possible partitioning requirements for achieving parallelism for a query operation, and (2) when the partitioning property of the data satisfies the partitioning requirements of a query operation. A data base management system in accordance with the invention uses parallel query processing techniques to optimize data repartitioning, or to avoid it altogether. These techniques apply to join operations, aggregation operations, and operations that apply subquery predicates. Unlike conventional query processing techniques, these techniques exploit the effect of data properties that arise from predicate application on the partitioning property of the data.

Other features and advantages of the present invention should be apparent from the following description of the preferred embodiment, which illustrates, by way of example, the principles of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a representation of horizontal partitioning of data tables in a data base management system constructed in accordance with the present invention.

FIG. **2** is a representation of a partitioned join operation in a data base management system with no data movement required.

FIG. **3** is a representation of a data base management system partitioned join operation that requires data movement.

FIG. **4** is a representation of a query execution plan (QEP) for a directed join operation in a data base management system.

FIG. **5** is a representation of a column homogenization process in a data base management system.

FIG. **6** is a representation of a build-dir-req process using "naive" processing, for generating a requirement for a source stream from a target partitioning requirement.

FIG. **7** is a representation of the build-dir-req process for building a partitioning requirement for the source stream of a join operation.

FIG. **8** is a representation of a build-subq-req process that is used in building a requirement for a local or directed subquery predicate application.

FIG. **9** is a representation of a build-dir-req process in accordance with the invention, which considers the effect of predicates binding partitioning key columns and considers correlation.

FIG. **10** is a representation of an improved version of the build-dir-req process in accordance with the invention, referred to as a listener-rp-pred process.

FIG. **11** is a representation of a local-agg-test process for determining when aggregation can be completed locally.

FIG. **12** is a representation of an improved version of the local-agg-test process illustrated in FIG. **11**, performed in accordance with the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

The present invention is implemented in a relational data base management system (RDBMS) whose operations are represented in the drawing figures. It should be understood that like reference numerals in the drawings refer to like elements.

Parallelism

Parallelism is often used in conjunction with a computer hardware architecture called shared-nothing to speed up the execution of queries. In a shared-nothing architecture, a collection of processors (or nodes) execute queries in parallel. A given query is broken up into subtasks, and all the subtasks are executed in parallel by the processors. Nodes in a shared-nothing architecture are typically connected by a high-speed communication network. The network is used to coordinate subtasks across nodes and also to exchange data between nodes. Each node has its own memory and disk.

To permit parallel query execution, tables are horizontally partitioned across nodes. Access to a given partition is only through the node which manages that partition. The rows of a table are typically assigned to a node by applying some deterministic partitioning function to a subset of the columns. These columns are called the partitioning key of the table. A simple example illustrating how a table might be partitioned is shown in FIG. **1**.

In FIG. **1**, Table A has been partitioned on a column A.x. Thus, the partitioning key is said to be A.x. The partitioning function assigns the rows of Table A to node **0** or to node **1**. In this example, rows with even values in A.x are assigned to node **0**, while rows with odd values in A.x are assigned to node **1**. Typically, the partitioning function is based on a simple hashing scheme.

Query Optimization in a Shared-Nothing RDBMS

The query optimizer in an RDBMS is responsible for translating an SQL query into an efficient query execution plan (QEP). The QEP dictates the methods and sequence used for accessing tables. These methods are used to join these tables, the placement of sorts, aggregation, predicate application, and so on. The QEP is interpreted by the database execution engine when the query is subsequently executed. There is added complexity in optimizing queries for a shared-nothing architecture. Among other things, the added complexity involves determining how to break the QEP into subtasks and then merge the results of subtasks for delivery to an end user.

In a shared-nothing architecture, the partitioning property is used during query optimization to keep track of how a table (or intermediate) result has been partitioned across nodes. The partitioning property describes two different aspects: (1) the nodes on which partitions of that table reside; and (2) the partitioning algorithm used to assign rows to nodes. Optimization choices about how queries are broken into subtasks are driven by the partitioning property. As an example, consider the query:

Query 1

```
select *
from A, B
where A.x = B.x
```

It should be noted that tables are referred to in queries by only their name, so that "A, B" is understood to refer to Table A and Table B. This convention will also be used occasionally in text.

If Tables A and B are partitioned over the same nodes, use the same partitioning algorithm, and are both partitioned on column x, then the join can be performed in parallel without any data movement. This is illustrated in FIG. **2**. As shown, no data movement is required to execute the join because the rows of A and B that satisfy the join predicate A.x=B.x are assigned to the same nodes. Parallel execution in this example is achieved by having a coordinator process, replicate, and ship identical QEPs joining Table A and Table B to node **0** and node **1**. Each node performs its join asynchronously and then returns rows back to the coordinator.

Partitioning Alternatives for Joins

The join in the previous example gives rise to a partitioning requirement. The join requires that Table A and Table B are partitioned over the same nodes, use the same partitioning function, and are partitioned on column x. That is, their partition property must be the same and they must be partitioned on the join column, x. If this was not the case, then it would have been necessary to dynamically repartition the tables to satisfy the partitioning requirement.

Repartitioning is illustrated using the same query as in the previous example. As before, assume Table A is partitioned on column x over nodes **0, 1** but now Table B is partitioned on some other column over nodes **0, 1, 2**. This is shown in FIG. **3**, which provides an example of a partitioned join operation that requires data movement.

In the situation represented in FIG. **3**, one or both of the tables needs to be dynamically repartitioned prior to executing the join. One option is to dynamically repartition Table B using the join column x among nodes **0,1** and to join this derived result with Table A in parallel. This join method is commonly known as a directed join because it directs rows of B to A. Alternatively, the query optimizer can replicate the data from Table A to all nodes of Table B. This join method is known as a broadcast join. It should be understood that the term "broadcast" will denote the replication of data across a set of nodes. Finally, the optimizer could decide to repartition both tables over some completely different set of nodes, again using the join column x to distribute rows. This parallel join method is typically called a repartition join. During optimization, a QEP would be generated for each of the different join methods mentioned above, assigned a cost based on an estimation of the QEP's execution time, and the least costly alternative would be chosen. The term "partitioned join" will be understood to refer to local, directed, or repartitioned join strategies. Unlike a broadcast join, these strategies serve to join partitions of both tables.

Partitioning Alternatives for Applying Subquery Predicates

Parallel execution strategies for subquery predicate application are analogous to the parallel join strategies described above. As noted above, implementing parallelism (that is, the parallelization decisions) for a join are hinged on how the two input tables are partitioned with respect to the join predicate(s). Likewise, the parallelization decisions for applying a subquery predicate hinge on how the table applying the subquery predicate and the table producing the subquery result are partitioned with respect to the subquery predicate(s). As an example, consider the query below:

Query 2

```
select *
from A
from A.x < > ALL (select B.x
    from B
    where B.y > 0)
```

As is in the case for a local join, if Tables A and B are partitioned over the same nodes, use the same partitioning algorithm, and are both partitioned on column x, then the subquery predicate can be applied in parallel without any data movement.

There are also directed, broadcast, and repartitioned versions of plans for parallel subquery predicate application. For example, if the subquery table were partitioned on something other than x, it could be directed by x values to the nodes of A. Similarly, the subquery result could be broadcast to the nodes of A. It is also possible to defer the application of the subquery predicate until after the table that applies the predicate has been repartitioned. Consequently, one could direct or broadcast the table applying the subquery to where the subquery result resides. Moreover, the optimizer could decide to repartition both tables over some completely different set of nodes prior to applying the subquery predicate. This strategy is congruous to a repartitioned join. The term "partitioned subquery predicate application" should be understood to refer to local, directed, or repartitioned subquery predicate application strategies.

Partitioning Alternatives for Aggregation

The parallel execution strategy for aggregation is typically carried out in two steps. Consider the following example Query 3.

Query 3

```
select count(*)
from A
group by y
```

Suppose that Table A is partitioned over multiple nodes using the partitioning key x. In the first step of query evaluation, aggregation is done on static partitions of A. Since A is not partitioned on the grouping column y, it is possible that rows in the same group reside on different nodes. Thus, the result of the first step of aggregation results in a partial count for each group on each node. In the final step, data is dynamically redistributed so that all rows with the same y value are on the same node. Partial counts are then added together to form the final result.

The final aggregation step is unnecessary if data is initially partitioned such that all rows of the same group are on one node. As described in the current literature, this is the case when the partitioning key columns are a subset of the grouping columns. It should be noted that the partitioning algorithm is deterministic. Two different rows will be assigned to the same node if they have the same input column values. Columns in the same group have the same values for the grouping columns. Thus, if the preceding example is changed so that Table A is initially partitioned on y, the final aggregation step is unnecessary.

Duplicate elimination (distinct) can be thought of as a special case of aggregation (with no aggregating functions); therefore, all of the parallelization decisions just described for aggregation apply to duplicate elimination as well.

The Effect of Other Properties on Partition Analysis

The term "partition analysis" will be used to designate the collection of processing involved in two tasks: (1) determining the partitioning requirements for a given operation; and (2) when a partitioning property satisfies a given partitioning requirement. In the preceding sections, basic partition analysis for simple queries involving joins, subquery predicate evaluation, and aggregation have been described. To perform this task efficiently, the optimizer has to deal with the effect of other properties to avoid unnecessary movement of data.

As an example of how other properties come into play, consider the following example Query 4:

Query 4

```
select *
from A, B
where A.x = 3 and A.y = B.y
```

Suppose that Table A is partitioned over multiple nodes using the composite partitioning key of columns A.x and A.y. Suppose that Table B is partitioned over the same nodes using column B.y.

As described above, the naive processing for determining the partitioning strategies for a join would fail to determine that a join strategy which directs tuples of B to nodes of A is a possibility. That is, they would fail to recognize that there is a way to repartition B on the nodes of A so that a local join could then take place between A and the result of repartitioning B. These naive processes require that there is a join predicate equating each of the corresponding columns of the partitioning key of A to some column of B. This criteria is not satisfied for A.x. A directed join strategy can be constructed, however, by taking advantage of the fact that A.x is bound to a constant value. This information can be used to repartition B using the values "3" and B.y and then do a local join with A and the repartition result.

Understanding of the rationale is best given via proof by contradiction. Suppose that B has been repartitioned using the values **3** and B.y. Suppose that the join of A and the repartitioned version of B cannot be done locally. Let k be a value of B.y from a row on node n_i which joins with some A row. Since the join cannot be done locally, there must exist some different node, n_j, where A.y=k. Observe that A.x=3 must be satisfied on all nodes. Thus, both rows from A have the same values for the partitioning columns, 3 and k. Since the partitioning algorithm is deterministic, n_i and n_j must be the same node.

Failure to recognize that the join can be done locally would result in at least one of the tables being repartitioned or broadcast needlessly. Such a plan might be orders of magnitude worse in performance. The present invention recognizes this.

Partitioning key columns can also be bound in a context-dependent way via correlated columns, as illustrated in the following example of Query 5:

Query 5

```
select *
from B
where B.z > ANY (select count(*)
      from A
      where x = B.x;
      group by y
```

Suppose that A is still partitioned as in the previous example. It was mentioned earlier that parallel aggregation requires two phases, unless the input to the initial aggregation phase is partitioned such that the partitioning key columns are a subset of the grouping columns. This condition does not hold here, because Table A is partitioned using columns A.x and A.y; nevertheless, the final aggregation step is not necessary, because A.x is bound to the correlation value B.x. The argument is similar to that provided for the previous example. The general effects of column equivalence and correlation must also be considered in partition analysis. This is described further below.

The term "partition analysis" has been defined to refer to algorithms for determining the partitioning requirements for QEP operations and for determining when a partitioning property satisfies these requirements. In particular, partition analysis has been described for join, subquery predicate evaluation, and aggregation queries. The effect of other properties on partition analysis also has been illustrated. Specifically, the way in which application of predicates effects partition analysis has been described. Failure to consider these effects can result in the unnecessary movement of data and a QEP, which may perform orders of magnitude worse than one that does consider these effects.

QEPs, Tuple Streams, and Properties

A query compiler for a relational data base management system (RDBMS) in accordance with the invention translates a high-level query into a query execution plan (QEP). This QEP is then interpreted by a database engine at

run-time. Conceptually, a QEP can be viewed as a dataflow graph of operators, where each node in the graph corresponds to a relational operation like join or a lower-level operation like sort. Each operator consumes one or more input sets of tuples (i.e., relations), and produces an output set of tuples (another relation). The input and output tuple sets will be referred to as tuple streams.

Each tuple stream includes a defining set of characteristics or properties \cite{Lohman88. Examples of properties include the set of tables accessed and joined to form the tuples in the stream, the set of columns that make up each tuple in the stream, the set of predicates that have been applied to the tuples in the stream, the partitioning of the tuples in the stream, and so forth.

Each operator in a QEP determines the properties of its output stream. The properties of an operator's output stream are a function of its input stream(s) and the operation being applied by the operator. For example, a sort operation (SORT operator) passes on all the properties of its input stream unchanged except for the order property. The repartition operation (RP operator) changes the partitioning property of the tuple stream. A query compiler will typically build a QEP bottom-up, operator-by-operator computing properties as it goes. Note that, depending on the implementation, some properties may be stored in QEP operators, while others may be recomputed when needed to save space.

As a QEP is built, partitioning requirements for new operators must be satisfied, e.g., the input streams of the join might need to be repartitioned or replicated so that they are compatible for a parallel join. The join operation first determines a set of target partitionings, and then for each, attempts to build a partitioning requirement for the join operands. If a join operand does not satisfy a partitioning requirement, an RP operator is added to the QEP for that operand. FIG. 4 shows an example of a directed join operation.

The outer table, Table A, is not initially partitioned on the join column, x. The inner table, Table B, is partitioned on the join column so the optimizer designates the inner table partitioning as the target partitioning. A partitioning requirement for the outer table is then derived from the target partitioning using the join predicates. The requirement consists of the nodes of the inner table and the join column. A request is made for an outer table QEP that satisfies this partitioning requirement. Because none exists, an RP operator is added to an existing QEP. Partitioning requirements and the RP operator will be described in more detail below.

With respect to join operations, other operators determine partitioning requirements based upon analysis of the query, and attempts to build a QEP which satisfies these requirements. For example, the aggregation operation (GROUP BY operator) generates a partitioning requirement for local aggregation and tests the partitioning property of the input stream to determine if a final aggregation step is necessary.

Relevant Properties Other than the Partitioning Property

Partition analysis is effected by more than the partitioning property. The relevant properties are briefly described below. The following description provides a general idea or references as to how relevant properties may be acquired or changed by different operators may be accomplished. The details of such operations should be apparent to those skilled in the art, in view of this disclosure. The description will focus on how each respective property is used in partition analysis.

One of the relevant properties is the tables property, which keeps track of the relations that have been accessed and joined to form the tuples in the stream. Different accesses to the same table are recorded as separate entries in the tables property. Another relevant property is the columns property. The columns of a tuple stream can include both columns from base relations and those derived from expressions. The columns property is used to keep track of a tuple stream's columns. Another property is the predicates property. During compilation, the predicates in a SQL query are typically broken down into conjunctive normal form. This allows each conjunct to be applied as an independent predicate. When a predicate (actually conjunct) is applied to an input tuple stream, each tuple in the resulting output stream has the property that it satisfies the predicate. The predicates property is used to keep track of all the predicates that have been applied to a tuple stream.

Another relevant property relates to column equivalence. For partition analysis, we are particularly interested in predicates that equate columns, like EMPNO=DEPTNO. These give rise to equivalence classes, which are sets of columns made equivalent by the application of equality predicates. Two columns can be made equivalent by a single predicate equating the two columns or by transitivity among two or more equality predicates. For a given equivalence class, one column is arbitrarily chosen as the equivalence class head. In the degenerate case, each column is equivalent to itself and thus in some equivalence class.

For partition analysis, it is assumed that, given a set of applied predicates, there is some way to determine: (1) whether some column C_i is in the same equivalence class as another column C_j; and (2) the equivalence class head for an arbitrary column C. It is important to point out that correlated columns are included in equivalence classes. For example, if the conjunct X=A is applied, where A is a correlated reference to another query block, then X and A are assumed to be in the same equivalence class.

Also considered are predicates equating columns of tables in an outer join operation in the same equivalence class for partition analysis. For example, if A.X=B.X is a predicate in an ON clause of an outer join operation between tables A and B, A.X and B.X are considered to be in the same equivalence class.

Bound Columns

Bound columns also will be considered in the following description. As illustrated above, partition analysis also needs to determine when a column has a constant value. A column C is bound in a tuple stream if C has the same value for every tuple in the stream. A column C can become bound in a variety of ways:

(a) C can be derived from a constant expression;

(b) there exists a predicate of the form C=10;

(c) a column in the equivalence class of C has become bound; or

(d) columns which functionally determine C have become bound

A column C can also become bound in a context-dependent way. For example, suppose we have C_i=C_j and C_j is a correlated column. Then C_i is bound in the context where C_j is correlated. For partition analysis, it is assumed that, given a set of applied predicates, there is some way to determine if a column C in a tuple stream is bound and whether this property is context-dependent or not.

Correlated Columns

In contrast to the columns property, which essentially keeps track of columns which are flowed from base tables,

the correlated columns property keeps track of column values which are supplied by some other tuple stream.

As an example, consider a join between two tables, Table T and Table R, using the join predicate T.C=R.C. Typically, an optimizer pushes down a join predicate so that it is applied when the inner table is accessed. Suppose that R is the inner table of a join and that the predicate has been pushed down to the access of R. Suppose further that R is accessed with an index scan ho as represented by an ISCAN operator. The ISCAN operator would apply T.C=R.C and include T.C in the correlated columns property to indicate that this value must be supplied by some other stream. When the tuple streams for R and T are finally joined, the join operator will remove T.C from the correlated columns property to indicate that the value is no longer needed as a correlation.

In addition to correlated column references which occur due to the optimizer's decision to push down a join predicate, correlated column references can occur because a subquery references a column as an outer-reference. For partition analysis, it is assumed that, given a set of applied predicates, and a set of tables to which the predicates will be applied, there is some way to determine the correlated column references. This is essentially computed by subtracting the columns that can be supplied by the tables from those referenced in the predicates.

The Partitioning Property

The partitioning property of a tuple stream represents how the stream's tuples are distributed among nodes of the system. The partitioning property identifies: (1) the set of nodes that may contain tuples of the stream; and (2) the partitioning algorithm used for assigning tuples to nodes.

A "nodegroup" is defined as a subset of the parallel system's nodes over which a tuple steam can be partitioned. Different streams may be distributed over the same nodegroup. Each nodegroup is assigned a unique identifier and this identifier is what is recorded in the partitioning property. Streams distributed over the same nodegroup, i.e. tables having the same nodegroup identifier, are said to be collocated.

A partitioning algorithm is needed only for multi-node nodegroups. The partitioning algorithm is typically implemented with a deterministic function, like a hash function, applied to some subset of the columns of the tuple stream. These columns are called the partitioning key. Since the function is deterministic, tuples with the same values for the partitioning key are assigned to the same node. The partitioning key is represented by listing the columns to which the partitioning function is applied. For example, (C_1, C_2, C_3) is a representation of a partitioning key. It is assumed that the order of the columns in the partitioning key is relevant. Therefore, saying that the tuple stream is partitioned on (C_1, C_2) is different than saying that it is partitioned on (C_2, C_1).

Specifically, the partitioning property is represented by three elements:

(1) the nodegroup identifier;

(2) the partitioning function identifier; and

(3) the partitioning key.

The latter two partitioning property elements are recorded only if the nodegroup contains multiple nodes. In certain cases, like in the case of a broadcast join, we may need to replicate tuples over all nodes in the nodegroup. The resulting partitioning property is represented with the nodegroup identifier and a special partitioning function which indicates that tuples are replicated. Thus, in this case, only the nodegroup identifier and partitioning function identifier are needed.

Finally, two partitioning properties are said to be equivalent if they are collocated, use the same partitioning function, have the same number of partitioning columns in the partitioning key, and corresponding partitioning key columns are in the same equivalence class. Single node nodegroups are equivalent if they are collocated. Inherent in this definition is the assumption that the partitioning function behaves the same regardless of the data types of its input. That is, if the predicate c_i=c_j evaluates to true, then it is assumed that the partitioning function produces the same value regardless of whether the value of c_i or c_j is used. This simplifies the exposition without loss of generality.

Partitioning Requirements

A partitioning requirement indicates a desired partitioning property for a QEP operation; consequently, a partitioning requirement has all of the same information included in a partitioning property. In general, it is assumed that a partitioning property and partitioning requirement are interchangeable and that one can be derived from the other via a simple cast function (a function that simply changes the data type). It is said that a partitioning property satisfies a partitioning requirement if the two are equivalent after the partitioning requirement has been cast to a property. If no QEP satisfying a partitioning requirement exists, an RP operator is added to one or more existing QEP's. The partitioning requirement is passed as an argument to the RP operator.

The RP Operator

The RP operator is the only operator which can change the partitioning property of the data. One of its arguments is a partitioning requirement which must be satisfied. It achieves this by applying the partitioning algorithm specified by the partitioning requirement to each tuple of the input stream and sending the tuple to the node (or nodes in the case of a broadcast algorithm) which results. The nodes of the input stream to which the RP operator applies the partitioning algorithm are called the producer nodes, and the nodes which receive the tuples of the producer nodes are called the consumer nodes. It is assumed that all producer and consumer nodes are directly connected.

When an RP operator is applied to a tuple stream containing correlated column references, data must flow both ways across connections. Correlated column values must first be sent from the consumer nodes to the producer nodes. The producer nodes then use the correlation values in their computations and send qualifying rows back to the consumer. This type of RP operator is called a listener RP. It is assumed that the node sending the correlation values, i.e. the node for which the producer is working on behalf of, is available on the producer side. The producer can use this node number to send rows back to the consumer. Alternatively, the consumer could supply its partitioning key columns and partitioning function to the producer. The producer could then apply the partitioning function to determine the node that will receive data. The latter approach is not generally as efficient since it might require the producer to send columns which were not even referenced as correlations by the producer. In accordance with the present invention, listener RP operations are minimized, since the

synchronization involved reduces parallelism. When a listener is unavoidable, the data flow is minimized by essentially pushing predicates which would be applied on the consumer side of an RP operation onto the producer side of the operation.

Column Homogenization

The column homogenization algorithm is used to map columns of requirements (such as order, key, or partitioning requirements) that are written in terms of columns of one set of tables to equivalent columns of another set of tables \cite{Simmen96. It maps a column to an equivalent column, subject to the constraint that the equivalent column belongs to one of a set of target tables. It takes as input a column, a set of predicates, and the target tables. The set of predicates defines the equivalence classes which are in effect.

To illustrate the idea, suppose we have a partitioning column A.X that we want to homogenize to the set of tables {B, C} using predicate conjuncts A.X=B.X and A.X=C.X. The equivalence class for A.X contains A.X, B.X, and C.X. There are two columns in the equivalence class which belong to one of the target tables; therefore, B.X or C.X is returned arbitrarily. Changing the example slightly, suppose we have the same column and predicates but the target tables are {D, E}. In this case, there is no equivalent column which maps to the set of target tables so NULL is returned.

FIG. **5** is pseudo-code that illustrates the processing performed by the column homogenization process. The process searches the equivalence class and returns the first column belonging to the target tables. If no equivalent column satisfying the constraint exists, then NULL is returned. If there are multiple possibilities for mapping, or homogenizing, the column, then one is returned arbitrarily. Note that a column is equivalent to itself and thus can homogenize to itself if the table it belongs to is in the set of target tables.

A "Naive" Algorithm for Generating a Requirement for a Partitioned Join or Subquery Predicate Application

A partitioned join or partitioned subquery predicate application requires that participating streams be equivalently partitioned. There can be multiple such partitionings. One approach at coming up with a target partitioning requirement for the operation is to cast the partitioning property of a QEP of one of the streams to a requirement. The stream providing this partitioning is called the target stream. Any QEP's whose partitioning property is broadcasting data are not permitted to provide the target partitioning. An equivalent partitioning requirement for the other stream is built, which is called the source stream, by homogenizing each of the partitioning key columns of the target partitioning requirement to columns of the source stream.

The build-dir-req process is illustrated in the pseudo-code of FIG. **6** for generating a requirement for a source stream from a target partitioning requirement.

The result partitioning is formed by using the nodegroup and partitioning function identifiers of the target partitioning requirement. The partitioning key of the result is formed by homogenizing each of the columns of the target partitioning key to a column of one of the source tables using the equivalence classes defined by the input predicates. If any column cannot be homogenized to a column of a source table, then the NULL partitioning requirement is returned, indicating that the join or subquery cannot be done in a partitioned fashion.

Using the build-dir-req Process to Build a Requirement for a Local or Directed Join

Next, consider the use of the build-dir-req process for building a partitioning requirement for the source stream of a join. The build-join-req process represented in the pseudo-code of FIG. **7** illustrates the processing steps performed.

This process takes a QEP for one operand of a join, the tables requirement for the source stream (the other join operand), and the join predicates. The partitioning property of the target QEP provides the target partitioning requirement for the join. The build-join-req process returns the partitioning requirement which must be satisfied by a QEP of the source stream. If the target partitioning requirement can be mapped to the source stream via the build-dir-req process, then it will return that; otherwise, it will return a partitioning requirement which broadcasts to node of the target stream. Thus, the build-join-req process favors a local or directed join over a broadcast join. The requirement returned by build-dir-req will trigger a local join if there is a source QEP with an equivalent partitioning property and a directed join if an RP operator must be added to an existing QEP to satisfy the partitioning requirement.

Consider the following simple example of Query 6:

Query 6

```
select *
from A, B
where A.x = B.x and A.y = B.y
```

Suppose that A is partitioned over multiple nodes represented by the nodegroup identifier 100, function identifier 100, and partitioning key (A.x, A.y). The build-join-req process is called with a QEP for A, the source table B, and the join predicates A.x=B.x and A.y=B.y. The build-dir-req process is able to return a requirement for a local or directed join by mapping the partitioning key columns to the source table. The resulting source requirement has a nodegroup identifier of **100**, a function identifier of **100**, and a partitioning key of (B.x, B.y). If a QEP for B has an equivalent partitioning property, then the join can be done locally; otherwise, an RP operator is added to an existing QEP.

Note that it is not necessary to derive the target partitioning requirement from the partitioning property of an existing QEP. One could start with any partitioning requirement as a target and then attempt to map it to both operands of the join using the build-dir-req process. The build-join-req process shows just one application of the build-dir-req process. The build-join-req process assumes that the transitive closure of the set of predicates has been completed; however, conventional naive algorithms say nothing about including correlation predicates which span the source and target tables. They also mention nothing about join predicates which span preserved and null-producing sides of an outer join. Moreover, they do not take advantage of predicates which bind columns to constant values or nor do they handle correlation in a general way.

Using the build-dir-req Process to Build a Requirement for a Local or Directed Subquery Predicate Application

The build-dir-req process is also used for building a requirement for a partitioned subquery predicate application. The build-subq-req process represented in FIG. **8** illustrates

its use in building a requirement for a local or directed subquery predicate application.

As in the build-join-req process, the partitioning property of the target QEP provides the target partitioning requirement. The build-subq-req process returns the partitioning requirement which must be satisfied by a QEP of the source stream. If the target partitioning requirement can be mapped to the source stream via build-dir-req, then it will return that; otherwise, it will return a partitioning requirement which broadcasts to the target stream.

The primary difference between the build-subq-req process and the build-join-req process is in the set of predicates supplied to build-dir-req for homogenizing the partitioning key. In the case of the join, build-join-req uses only conjuncts applied by the join operand. When generating the subquery requirement, however, the build-subq-req process will use correlation predicates, and in certain cases, it can also use the subquery predicate (even when it is not a conjunct).

Consider the following example of Query 7:

Query 7

```
select *
from A
        where P or A.y = ANY (select B.z
        from B
        where B.x = A.x);
```

Suppose that A provides the target partitioning for a parallel subquery predicate evaluation and that it is again partitioned over multiple nodes represented by nodegroup identifier 100 and function identifier 100 applied to partitioning key (A.x, A.y).

The build-subq-req process is called with the correlation predicate B.x=A.x. It then adds the subquery predicate A.y=B.z and calls the build-dir-req process in an attempt to return a source partitioning requirement for a local or directed subquery predicate application. Using the correlation and subquery predicates, the build-dir-req process returns a source requirement with a nodegroup identifier of 100, a function identifier of 100, and a partitioning key (B.x, B.z). If a QEP for B has an equivalent partitioning property then the join can be done locally; otherwise, an RP operator is added to an existing QEP.

An Improved Process for Generating a Requirement for a Partitioned Join or Subquery Predicate Application

When the build-dir-req process fails to determine that an equivalent partitioning requirement exists for the source stream, then either a broadcast join or broadcast subquery predicate application results. This is clearly less efficient than a local join or local subquery predicate application. It is also typically less efficient than one which directs tuples to the appropriate node.

The build-dir-req process described in the previous section maps a target partitioning to a source stream given a set of predicates for mapping between the target and source streams. The algorithm generally mimics what is in the open literature, and as was illustrated earlier, fails to consider the effect of predicates binding partitioning key columns. It also fails to handle correlation in a general way. The improved build-dir-req process shown in FIG. **9** remedies these shortcomings.

Like the naive algorithm, the improved build-dir-req process takes a target partitioning requirement, the tables of the source stream, and a set of predicates for mapping the partitioning key of the target partitioning requirement to columns of the source stream. The general idea is the same. The source partitioning is formed by using the nodegroup identifier and partitioning function identifier of the target partitioning. The partitioning key is formed by mapping the partitioning key column to the columns of the source stream.

There are two key differences between the naive and improved versions of the algorithm:

(1) A partitioning key column of the target partitioning requirement which is bound to a constant is considered mapped to the source stream by the constant; and

(2) A partitioning key column of the target partitioning requirement which is referenced as a correlation by one of the source tables is considered mapped to the source stream by the correlation.

The effect of each of these changes will be illustrated by considering again the example of Query 4, described above:

Query 4

```
select *
from A, B
where A.x = 3 and A.y = B.y
```

Suppose that Table A is partitioned over multiple nodes represented by nodegroup identifier 100, function identifier 100, and partitioning key (A.x, A.y). Using the build-join-req process that calls the earlier version of build-dir-req will result in the return of a source requirement that broadcasts to nodegroup 100. The reason is that the earlier version of build-dir-req fails to homogenize partitioning key column A.x, since there is no join predicate mapping it to a column of the source table B; consequently, the NULL partitioning requirement is returned.

The novel version of the build-dir-req process is able to generate a source partitioning requirement for a local or directed join. It is able to recognize that A.x is bound to the constant value 3 and maps it to the source stream using the constant. Using the join predicate to map the other partitioning key column, A.y, the source partitioning key for the join is (3, B.y). An RP operator which directs via partitioning key (3, B.y) can be added to an existing QEP for the source stream.

Consider next the change to the build-dir-req process that deals with correlation.

Query 8

```
select *
from A
where P or A.y = ANY (select B.z)
        from B
        where B.x > A.x;
```

Suppose that Table A provides the target partitioning for a parallel subquery predicate evaluation and that it is again partitioned over multiple nodes represented by nodegroup identifier 100 and function identifier 100 applied to partitioning key (A.x, A.y).

Using the build-subq-req algorithm which calls the earlier version of build-dir-req will result in the return of a source requirement which broadcasts to nodegroup **100**. Again, the

reason is that the earlier version of build-dir-req fails to homogenize partitioning key column A.x since there is no join predicate mapping it to a column of the source table B; consequently, the NULL partitioning requirement is returned. The new version of build-subq-req is able to generate a source partitioning requirement for a local or directed subquery predicate application. It is able to recognize that A.x is referenced as a correlation and maps it to the source stream as is. Using the subquery predicate to map the other partitioning key column, A.y, the source partitioning key for the subquery is (A.x, B.z).

Thus, an RP operator which directs rows via partitioning key (A.x, B.z) can be added to an existing QEP producing the subquery result.

Changes to the Listener-RP Operator

Note that the RP operator added in the previous example would be a listener RP since the stream that the RP operator is applied to has a correlated column reference, A.x. Recall that a listener RP operator receives the correlation values from a consumer node, or caller, uses them in its computations, and sends qualifying tuples back to the caller. It was indicated that the caller's node number is available to the producer and that the RP operator simply uses it to determine where to send qualifying tuples as opposed to having the caller pass its partitioning key values. A change can be made to the behavior of the RP operator which will take advantage of the improvements to build-dir-req. The listener-rp-pred process represented in FIG. **10** illustrates these changes.

FIG. **10** describes a predicate which is applied to a tuple that the listener RP operator is getting ready to send back to the caller. If the RP operator has a target partitioning requirement with a partitioning key, it first applies the partitioning function to these column values. If the node-number is the same as that of the caller, then the tuple is sent back to the caller; otherwise, the tuple is not sent back. The effect of this change, in conjunction with the changes to build-dir-req, is to apply (either join or subquery) predicates which will be applied on the consumer side of an RP operator on the producer side.

Consider the previous example of Query 8 again.

Query 8

```
select *
  from A
where P or A.y = ANY (select B.z)
        from B
        where B.x > A.x;
```

Using the improved build-dir-req process, a source requirement was determined for directing records from the stream producing the subquery result to the stream applying the subquery predicate. The partitioning key of this requirement is (A.x, B.z). Using the predicate described above, the listener RP operator could avoid sending many rows that would not satisfy the subquery predicate A.y=B.z.

To illustrate this further, assume that a tuple on some node N of A has the values 1 for x and 1 for y and a tuple on a different node of B has the values 10 for x and 2 for z. Note that the B tuple satisfies the subquery predicate B.x>A.x. Therefore, if the listener RP operator did not apply the predicate described by listener-rp-pred, it would send the record back to node N. However, this tuple will not satisfy the subquery predicate A.y=B.z. Applying listener-rp-pred to the tuple would reduce the likelihood of this happening. It would apply the partitioning function to partitioning key values A.x=1 and B.z=2 and compare the result to the caller's node. Assuming that tuples of table A having x=1 and y=2 and that tuples having x=1 and y=1 are assigned to different nodes, this predicate would then avoid sending a tuple which could not satisfy the subquery predicate. It is possible that another tuple on node N of Table A has and x=1 and y=2. This is why the subquery predicate is still applied on the consumer side. This change offers some early filtering in cases where this is not true. Note that if all of the columns in the result of build-dir-req are correlated, then the RP operator can avoid applying the listener-rp-pred process, since it will do no filtering. This explains why build-dir-req returns the NULL partition requirement when none of the partitioning key columns of the target map to source columns.

Using the Improved build-dir-req Process to Build a Requirement for a Local or Directed Join

Some changes are implemented to the conventional build-join-req process. First of all, it will call the improved version of build-dir-req which exploits the binding of partitioning key columns to constants and their references as correlations. Consequently, in addition to join predicates, the process will be called with local predicates and predicates referenced in the source stream which are correlated to the target stream. Outer join predicates are permitted as well.

Using the Improved build-dir-req Process to Build a Requirement for a Local or Directed Subquery Predicate Application

A new version of the build-subq-req process is not implemented, but changes to the conventional process will use the improved version of build-dir-req and that it will also supply local predicates as well as predicates referenced in the source stream which are correlated to the target stream. We also note a change with regard to which subquery predicates can supply a predicate to build-dir-req. A subquery predicate of the form column 1<>ALL column 2 is allowed to contribute to the mapping predicates and is passed to build-dir-req. This is semantically incorrect if one of the columns where to have a null value. It is noted that build-subq-req should exclude this predicate unless both columns are not nullable.

A Naive Algorithm for Determining If Aggregation Can be Completed Locally

It was noted that the parallel execution strategy for aggregation begins by aggregating on each partition of the input stream. Then, if necessary, data is repartitioned so that tuples in the same group land on the same node and a final aggregation step takes place. It was noted that the final aggregation step can be avoided if data is initially partitioned so that tuples in the same group are on same node. This is the case when the partitioning key columns of the input stream are a subset of the grouping columns.

The local-agg-test process shown in FIG. **11** is the currently known test for determining when aggregation can be completed locally.

The process tests the partitioning property of the input QEP against the grouping columns and returns true if either (a) the QEP is on a single node; or (b) the QEP's partitioning key is a subset of the grouping columns. The process returns false otherwise.

Note that the process takes column equivalence classes into account as shown in the following example of Query 9.

Query 9

```
select count(*)
from A
where x = z
group by z,y
```

Suppose that Table A is partitioned over multiple nodes using the partitioning key (x). After substituting the grouping column, x, with the equivalent column, z, it is clear that the partitioning key columns are a subset of the grouping columns. The effect of column equivalence classes is taken into account on line 11 where it considers a partitioning key column a member of the grouping columns if it is in the same equivalence class as a grouping column. The algorithm does not, however, take into account the effect of predicates which bind partitioning key columns to constant or correlated values. Failure to take these effects into account could result in unnecessary repartitioning of data.

An Improved Process for Determining If Aggregation Can be Completed Locally

The improved version of the local-agg-test process is shown in FIG. 12.

The improved version makes a simple yet powerful change to the previous version of the algorithm. It assumes that any partitioning key column which is bound to a constant or correlation value is vacuously a member of the grouping columns. In effect, we now say that aggregation can be completed locally if the unbound partitioning key columns are a subset of the grouping columns.

Consider the example of Query 10, where this change prevents repartitioning.

Query 10

```
select count(*)
from A
where x = 3
group by y
```

Suppose that Table A is partitioned over multiple nodes using partitioning key (x, y). It should be noted that the partitioning key columns (x, y) are not a subset of the grouping columns (y). Aggregation can be completed locally, however, since partitioning key column, x, is bound to a constant value. The proof is best given by contradiction. The argument is similar to that given above, for the example that made use of constants to facilitate a directed join operation.

Finally, note that if all of the partitioning key columns are bound to a constant or correlation value, then the algorithm returns true regardless of what the grouping columns are. In fact, it will return true in this case if there are no grouping columns as illustrated by the following example of Query 11.

Query 11

```
select *
from B
where B.z = (select count(*)
from A
where x = B.x and y = 3)
```

Again, suppose that A is partitioned over multiple nodes using partitioning key (x, y). Note that x is bound to a correlation value and y is bound to a constant. Thus, each is vacuously assumed to be contained in the set of grouping columns, even though there are none. Note that these bindings effectively limit each execution of the subquery to a single node.

Advantages of the Invention

This disclosure described novel techniques used in performing partition analysis during query optimization for joins, subquery predicate application, and aggregation. One of the main goals of partition analysis is to optimize or avoid data repartitioning by recognizing the possible partitioning requirements for achieving parallelism for a query operation and when the partitioning property of the data satisfies the partitioning requirements of a query operation. Unlike known techniques, the techniques presented here consider the effect of predicates during partition analysis.

An algorithm called build-dir-req was presented above for generating a partitioning requirement for a partitioned join or partition subquery predicate application. Its use in determining a requirement for a local or directed join was shown by the description of the build-join-req process. Also illustrated was its use in determining a requirement for a local or directed subquery predicate application via the build-subq-req algorithm.

The build-dir-req process is an improvement over the current state of the art in that it takes into account predicates that bind partitioning key columns to constant values. It also handles partitioning key columns which are referenced as correlations in a general way. The latter changes sometimes make it possible to filter tuples on the producer side of a listener RP operation which would not satisfy predicates which will be applied on the consumer side. A listener-rp-pred process that takes advantage of this was described.

Finally, a process called local-agg-test was described for determining when aggregation can be completed locally. Unlike known techniques, this process takes into account predicates which bind partitioning key columns to constant or correlated values.

The present invention has been described above in terms of a presently preferred embodiment so that an understanding of the present invention can be conveyed. There are, however, many configurations for SQL-processing relational data base management systems not specifically described herein but with which the present invention is applicable. The present invention should therefore not be seen as limited to the particular embodiments described herein, but rather, it should be understood that the present invention has wide applicability with respect to SQL-processing relational data base management systems generally. All modifications, variations, or equivalent arrangements and implementations that are within the scope of the attached claims should therefore be considered within the scope of the invention.

What is claimed is:

1. A method of processing a query in a relational database management system that operates in a computer network to retrieve data from tables in computer storage, the method comprising the steps of:

receiving two partitioned data streams that relate to a join operation;

determining whether conjunct predicates can be used to locally perform a parallel inner join or outer join; and

performing the join operation locally if the step of determining indicates appropriate conjunct predicates can be used.

2. A method as defined in claim 1, wherein the step of determining comprises applying conjunct predicates that equate columns of the same table.

3. A method as defined in claim 1, wherein the step of determining comprises applying conjunct predicates that equate two columns, where one column is bound.

4. A method as defined in claim 1, wherein the step of determining comprises applying conjunct predicates that equate columns of two different tables.

5. A method as defined in claim 4, wherein neither one of the columns referenced by the column-equating predicates comprises a correlated column reference.

6. A method as defined in claim 4, wherein one of the columns referenced by the column-equating predicates comprises a correlated column reference.

7. A method as defined in claim 1, wherein the partitioned data streams comprise a source data stream and a target data stream, and the method further comprises the steps of:

receiving a target query evaluation plan (QEP) that produces tuples of the target data stream, which are to be joined with the source data stream;

receiving a source tables requirement for the tables of the source data stream to be joined;

receiving a set of source QEPs that produce tuples of the source data stream;

receiving a set of conjunct predicates determined to be suitable for locally performing a parallel inner join or outer join;

generating a source partitioning requirement from a partitioning property of the target QEP, the source tables requirement, and the set of conjunct predicates; and

indicating that a local join is possible if there is a source QEP with a partitioning property that is equivalent to the source partitioning requirement.

8. A method as defined in claim 7, wherein source partitioning requirements generated by the step of determining are generated by performing a build-dir-req process, wherein the build-dir-req process comprises the steps of:

setting a node group identifier of the source partitioning requirement to a node group identifier of the partitioning property of the target QEP;

setting a partitioning function identifier of the source partitioning requirement to that of a partitioning function identifier of the partitioning property of the target QEP;

forming the partitioning key of the source partitioning requirement by mapping partitioning key columns of the partitioning property of the target QEP to a column of the source data stream, wherein:

a column of the partitioning key of the target QEP that is bound to a constant value C in the target data stream is mapped to the source data stream as C;

a column of the partitioning key of the target QEP that is bound to a correlated value C in the target data stream is mapped to the source data stream as C; and

a column of the partitioning key of the target QEP that is in an equivalence class the same as a column C of the source data stream is mapped as C.

9. A method of processing a query in a relational database management system that operates in a computer network to retrieve data from tables in computer storage, the method comprising the steps of:

receiving two partitioned data streams that comprise a source data stream and a target data stream, wherein the data streams relate to a join operation;

determining whether conjunct predicates can be used to direct tuples of the source data stream to the target data stream; and

performing the join operation after directing the tuples of the source data stream to the target data stream if the step of determining indicates appropriate conjunct predicates can be used.

10. A method as defined in claim 9, wherein the step of determining comprises applying conjunct predicates that equate columns of the same table.

11. A method as defined in claim 9, wherein the step of determining comprises applying conjunct predicates that equate two columns, where one column is bound.

12. A method as defined in claim 9, wherein the step of determining comprises applying conjunct predicates that equate columns of two different tables.

13. A method as defined in claim 12, wherein neither one of the columns referenced by the column-equating predicates comprises a correlated column reference.

14. A method as defined in claim 12, wherein one of the columns referenced by the column-equating predicates comprises a correlated column reference.

15. A method as defined in claim 9, wherein the method further comprises the steps of:

receiving a target query evaluation plan (QEP) that produces tuples of the target data stream, which are to be joined with the source data stream;

receiving a source tables requirement for the tables of the source data stream to be joined;

receiving a set of conjunct predicates determined to be suitable for determining if the join can be done by directing tuples from nodes of the source data stream to nodes of the target data stream;

generating a source partitioning requirement from a partitioning property of the target QEP, the source tables requirement, and the set of conjunct predicates; and

indicating that the join operation can be performed by directing tuples from nodes of the source data stream to nodes of the target data stream, if a source partitioning is generated, by returning the source partitioning requirement, otherwise indicating that the join operation cannot be directed, by returning a broadcast partitioning requirement.

16. A method as defined in claim 15, wherein source partitioning requirements generated by the step of determining are generated by performing a build-dir-req process, wherein the build-dir-req process comprises the steps of:

setting a node group identifier of the source partitioning requirement to a node group identifier of the partitioning property of the target QEP;

setting a partitioning function identifier of the source partitioning requirement to that of a partitioning function identifier of the partitioning property of the target QEP;

forming the partitioning key of the source partitioning requirement by mapping partitioning key columns of

the partitioning property of the target QEP to a column of the source data stream, wherein:

a column of the partitioning key of the target QEP that is bound to a constant value C in the target data stream is mapped to the source data stream as C;

a column of the partitioning key of the target QEP that is bound to a correlated value C in the target data stream is mapped to the source data stream as C; and

a column of the partitioning key of the target QEP that is in an equivalence class the same as a column C of the source data stream is mapped as C.

17. A method of processing a query in a relational database management system that operates in a computer network to retrieve data from tables in computer storage, the method comprising the steps of:

receiving two data streams that comprise a source data stream and a target data stream, wherein a subquery predicate will be applied to the data streams;

determining whether the subquery predicate can be applied to the target data stream locally in parallel; and

applying the determined subquery predicate in parallel, locally, if the step of determining indicates the subquery predicate can be so applied.

18. A method as defined in claim **17**, wherein the step of determining comprises applying conjunct predicates that equate columns of the same table.

19. A method as defined in claim **17**, wherein the step of determining comprises applying conjunct predicates that equate two columns, where one column is bound.

20. A method as defined in claim **17**, wherein the step of determining comprises applying conjunct predicates that equate columns of two different tables.

21. A method as defined in claim **18**, wherein neither one of the columns referenced by the column-equating predicates comprises a correlated column reference.

22. A method as defined in claim **18**, wherein one of the columns referenced by the column-equating predicates comprises a correlated column reference.

23. A method as defined in claim **15**, wherein the step of determining comprises applying a subquery predicate that equates a column of a table in the target data stream with a column in the source data stream, and a subquery operator is either an ANY operator or an IN operator.

24. A method as defined in claim **15**, wherein the step of determining comprises applying a subquery predicate that equates a non-nullable column of a table in the target data stream with a non-nullable column in the source data stream, and a subquery operator is either a NOT ALL operator or a NOT IN operator.

25. A method as defined in claim **15**, wherein the partitioned data streams comprise a source data stream and a target data stream, and the method further comprises the steps of:

receiving a target query evaluation plan (QEP) that produces tuples of the target data stream, which are to be compared with tuples of the source data stream by a subquery predicate;

receiving a source tables requirement for the tables of the data stream producing the subquery tuples;

receiving a set of conjunct predicates determined to be suitable for deciding whether the subquery predicate can be applied to the target data stream locally, in parallel;

generating a source partitioning requirement from a partitioning property of the target QEP, the source tables requirement, and the set of conjunct predicates; and

indicating that a local join is possible if there is a source QEP with a partitioning property that is equivalent to the source partitioning requirement.

26. A method as defined in claim **25**, wherein source partitioning requirements generated by the step of determining are generated by performing a build-dir-req process, wherein the build-dir-req process comprises the steps of:

setting a node group identifier of the source partitioning requirement to a node group identifier of the partitioning property of the target QEP;

setting a partitioning function identifier of the source partitioning requirement to that of a partitioning function identifier of the partitioning property of the target QEP;

forming the partitioning key of the source partitioning requirement by mapping partitioning key columns of the partitioning property of the target QEP to a column of the source data stream, wherein:

a column of the partitioning key of the target QEP that is bound to a constant value C in the target data stream is mapped to the source data stream as C;

a column of the partitioning key of the target QEP that is bound to a correlated value C in the target data stream is mapped to the source data stream as C; and

a column of the partitioning key of the target QEP that is in an equivalence class the same as a column C of the source data stream is mapped as C.

27. A method of processing a query in a relational database management system that operates in a computer network to retrieve data from tables in computer storage, the method comprising the steps of:

receiving two partitioned data streams that comprise a source data stream and a target data stream, wherein a subquery predicate will be applied to the data streams;

determining whether the subquery predicate and a conjunct predicate can be used to direct tuples of the source data stream to the target data stream; and

directing the tuples of the source data stream to the target data stream if the step of determining indicates appropriate predicates can be so used.

28. A method as defined in claim **27**, wherein the conjunct predicate in the step of determining comprises a conjunct predicate that equates columns of the same table.

29. A method as defined in claim **27**, wherein the conjunct predicate in the step of determining comprises a conjunct predicate that equates two columns, where one column is bound.

30. A method as defined in claim **27**, wherein the conjunct predicate in the step of determining comprises a conjunct predicate that equates columns of two different tables.

31. A method as defined in claim **30**, wherein neither one of the columns referenced by the column-equating predicates comprises a correlated column reference.

32. A method as defined in claim **30**, wherein one of the columns referenced by the column-equating predicates comprises a correlated column reference.

33. A method as defined in claim **27**, wherein the step of determining comprises applying a subquery predicate that equates a column of a table in the target data stream with a column in the source data stream, and a subquery operator is either an ANY operator or an IN operator.

34. A method as defined in claim **27**, wherein the step of determining comprises applying a subquery predicate that equates a non-nullable column of a table in the target data stream with a non-nullable column in the source data stream, and a subquery operator is either a NOT ALL operator or a NOT IN operator.

35. A method as defined in claim 27, wherein the method further comprises the steps of:

receiving a target query evaluation plan (QEP) that produces tuples of the target data stream, which are to be compared with tuples of the source data stream by a subquery predicate;

receiving a source tables requirement for the tables of the data stream producing the subquery tuples;

receiving a set of conjunct predicates determined to be suitable for deciding whether the subquery predicate can be applied to the target data stream by directing tuples from nodes of the source data stream to nodes of the target data stream;

generating a source partitioning requirement from a partitioning property of the target QEP, the source tables requirement, and the set of conjunct predicates; and

indicating that a local join is possible if there is a source QEP with a partitioning property that is equivalent to the source partitioning requirement, and otherwise indicating that the join operation cannot be directed and returning a broadcast partitioning requirement.

36. A method as defined in claim 35, wherein source partitioning requirements generated by the step of determining are generated by performing a build-dir-req process, wherein the build-dir-req process comprises the steps of:

setting a node group identifier of the source partitioning requirement to a node group identifier of the partitioning property of the target QEP;

setting a partitioning function identifier of the source partitioning requirement to that of a partitioning function identifier of the partitioning property of the target QEP;

forming the partitioning key of the source partitioning requirement by mapping partitioning key columns of the partitioning property of the target QEP to a column of the source data stream, wherein:

a column of the partitioning key of the target QEP that is bound to a constant value C in the target data stream is mapped to the source data stream as C;

a column of the partitioning key of the target QEP that is bound to a correlated value C in the target data stream is mapped to the source data stream as C; and

a column of the partitioning key of the target QEP that is in an equivalence class the same as a column C of the source data stream is mapped as C.

37. A method of processing a query in a relational database management system that operates in a computer network to retrieve data from tables in computer storage, the method comprising the steps of:

receiving a query that includes a relational operator that specifies one or more columns of tables in the computer storage that are distributed with a source partitioning across nodes of the computer network that are to be compared with one or more rows of tables in the computer storage that are distributed with a target partitioning across nodes of the computer network;

receiving a target partitioning requirement, tables of the source stream of data, and a set of predicates for mapping a partitioning key of the target partitioning requirement to columns of the source stream;

performing a repartitioning correlated RP operator of the system in response to a request from a consumer node to deliver a repartitioned table to a target node, wherein the RP operator comprises the steps of:

applying the source partitioning as a predicate operation to a tuple that the RP operator is to send back to the consumer node;

applying a partitioning function to column values if the RP operator is specified with the target partitioning requirement;

sending the tuple back to the consumer node if the target node of the RP operator is the same as that of the consumer node; and otherwise not sending the tuple back to the consumer node.

38. A method of processing a query in a relational database management system that operates in a computer network to retrieve data from tables in computer storage, the method comprising the steps of:

receiving a partitioned data stream to which a table operation comprising either an aggregation operation or a distinct operation will be applied;

determining whether conjunct predicates can be used to locally perform the table operation in parallel; and

performing the parallel table operation locally if the step of determining indicates appropriate conjunct predicates can be used.

39. A method as defined in claim 38, wherein the step of determining comprises applying conjunct predicates that equate columns of the same table.

40. A method as defined in claim 38, wherein the step of determining comprises applying conjunct predicates that equate two columns, where one column is bound.

41. A method as defined in claim 38, wherein the step of determining comprises applying conjunct predicates that equate columns of two different tables.

42. A method as defined in claim 41, wherein neither one of the columns referenced by the column-equating predicates comprises a correlated column reference.

43. A method as defined in claim 41, wherein one of the columns referenced by the column-equating predicates comprises a correlated column reference.

44. A method as defined in claim 38, further including the step of indicating that an aggregation operation can be performed locally, in parallel, if upon substituting columns of a key of the partitioning operation and columns of a key of the aggregating operation or of a key of the distinct operation with equivalence class heads, it is determined that unbound partitioning key columns are a subset of the aggregating or distinct key columns.

* * * * *

US006356892B1

(12) **United States Patent**
Corn et al.

(10) **Patent No.: US 6,356,892 B1**
(45) **Date of Patent: Mar. 12, 2002**

(54) **EFFICIENT IMPLEMENTATION OF LIGHTWEIGHT DIRECTORY ACCESS PROTOCOL (LDAP) SEARCH QUERIES WITH STRUCTURED QUERY LANGUAGE (SQL)**

(75) Inventors: **Cynthia Fleming Corn**; **Larry George Fichtner**; **Rodolfo Augusto Mancisidor**; **Shaw-Ben Shi**, all of Austin, TX (US)

(73) Assignee: **International Business Machines Corporation**, Armonk, NY (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/160,022**

(22) Filed: **Sep. 24, 1998**

(51) **Int. Cl.**[7] **G06F 17/30**
(52) **U.S. Cl.** **707/3**; 707/4; 707/103
(58) **Field of Search** 707/2, 4, 3, 104, 707/20; 709/217, 218, 219, 202

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,379,419	A	* 1/1995	Heffernan et al.	707/4
5,412,804	A	* 5/1995	Krishna	707/2
5,537,590	A	7/1996	Amado	
5,584,024	A	* 12/1996	Shwartz	707/4
5,600,831	A	* 2/1997	Levy et al.	707/2
5,668,987	A	9/1997	Schneider	
5,701,400	A	12/1997	Amado	
5,864,842	A	* 1/1999	Pederson et al.	707/3
5,907,837	A	* 5/1999	Ferrel et al.	707/3
5,956,706	A	* 9/1999	Carey et al.	707/2
5,963,933	A	* 10/1999	Cheng et al.	707/2
5,995,961	A	* 11/1999	Levy et al.	707/4
6,009,428	A	* 12/1999	Kleewein et al.	707/10
6,016,499	A	* 1/2000	Ferguson	707/104
6,052,681	A	* 4/2000	Harvey	707/3
6,055,562	A	* 4/2000	Devarakonda et al.	709/202
6,085,188	A	* 7/2000	Bachmann et al.	707/3
6,092,062	A	* 7/2000	Lohman et al.	707/2
6,199,062	B1	* 3/2001	Byrne et al.	707/3

* cited by examiner

Primary Examiner—Kim Vu
Assistant Examiner—Anh Ly
(74) *Attorney, Agent, or Firm*—Winstead Sechrest & Minick P.C.; Jeffrey S. LaBaw

(57) **ABSTRACT**

A method of hierarchical LDAP searching in an LDAP directory service having a relational database management system (DBMS) as a backing store. The method begins by parsing an LDAP filter-based query for elements and logical operators of the filter query. For each filter element, the method generates an SQL subquery according to a set of translation rules. For each SQL subquery, the method then generates a set of entry identifiers for the LDAP filter query. Then, the SQL subqueries are combined into a single SQL query according to a set of combination rules chosen corresponding to the logical operators of the LDAP filter query.

29 Claims, 5 Drawing Sheets

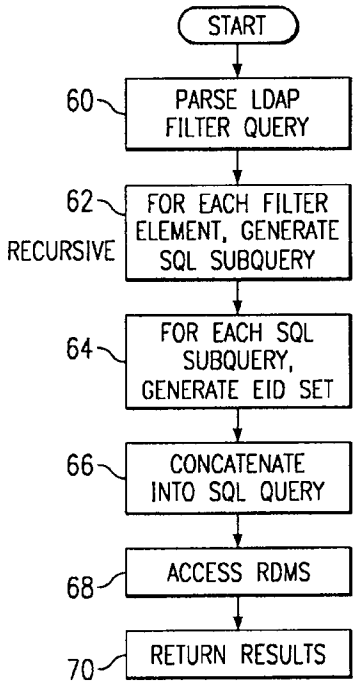


FIG. 1

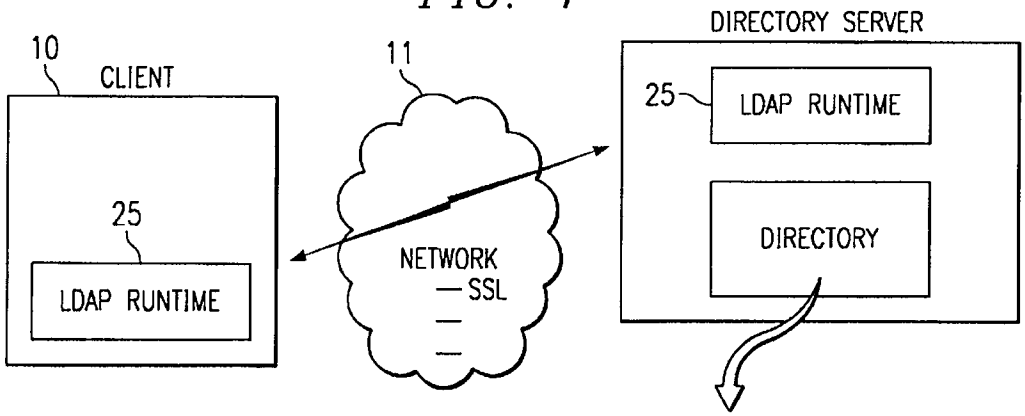
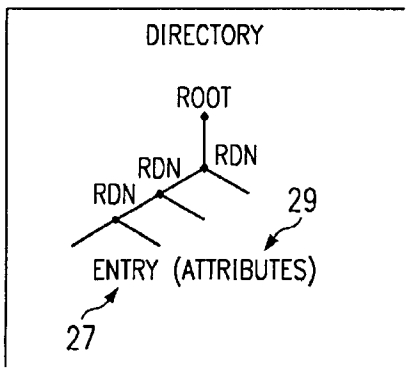


FIG. 2



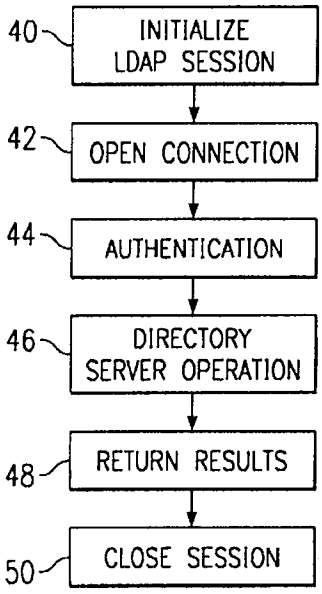


FIG. 3

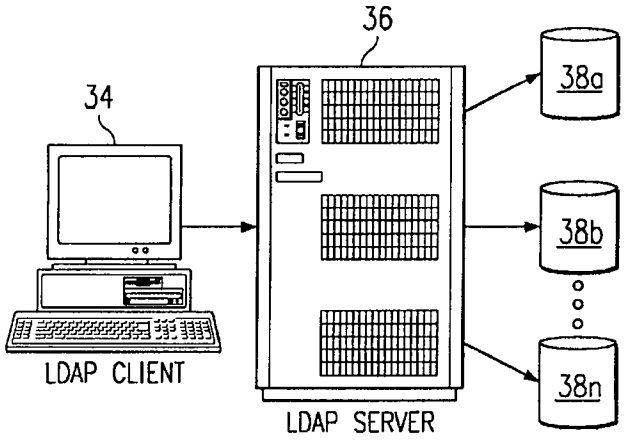


FIG. 4A

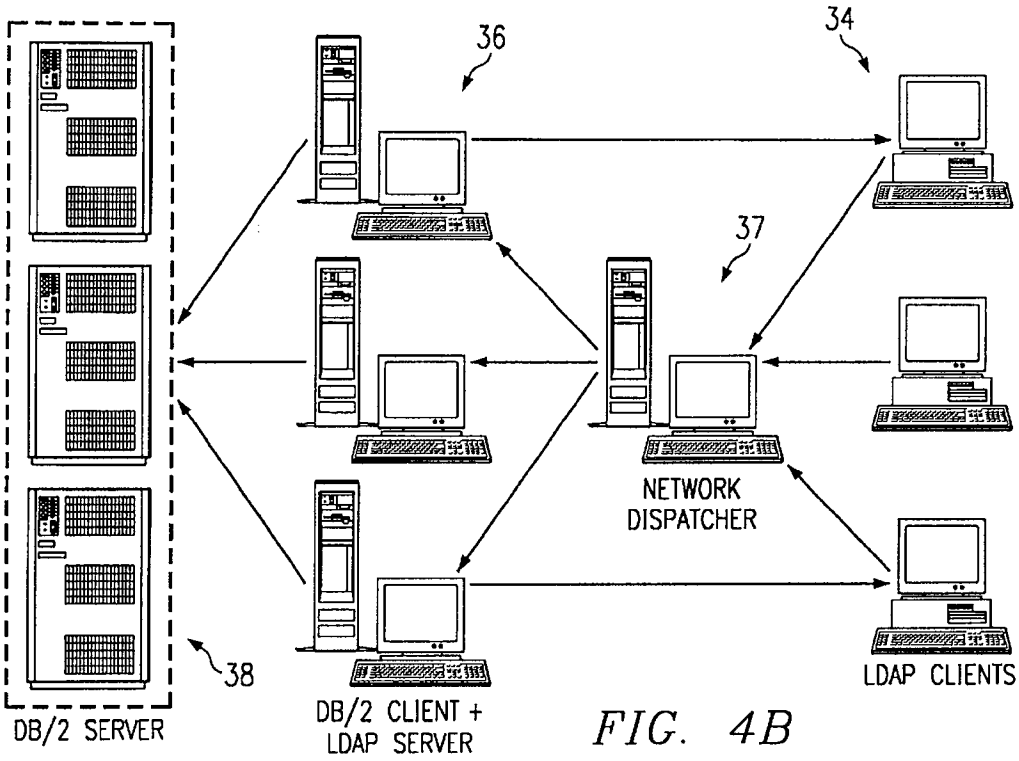

36
34
37
NETWORK
DISPATCHER
38
DB/2 SERVER
DB/2 CLIENT +
LDAP SERVER
LDAP CLIENTS

FIG. 4B

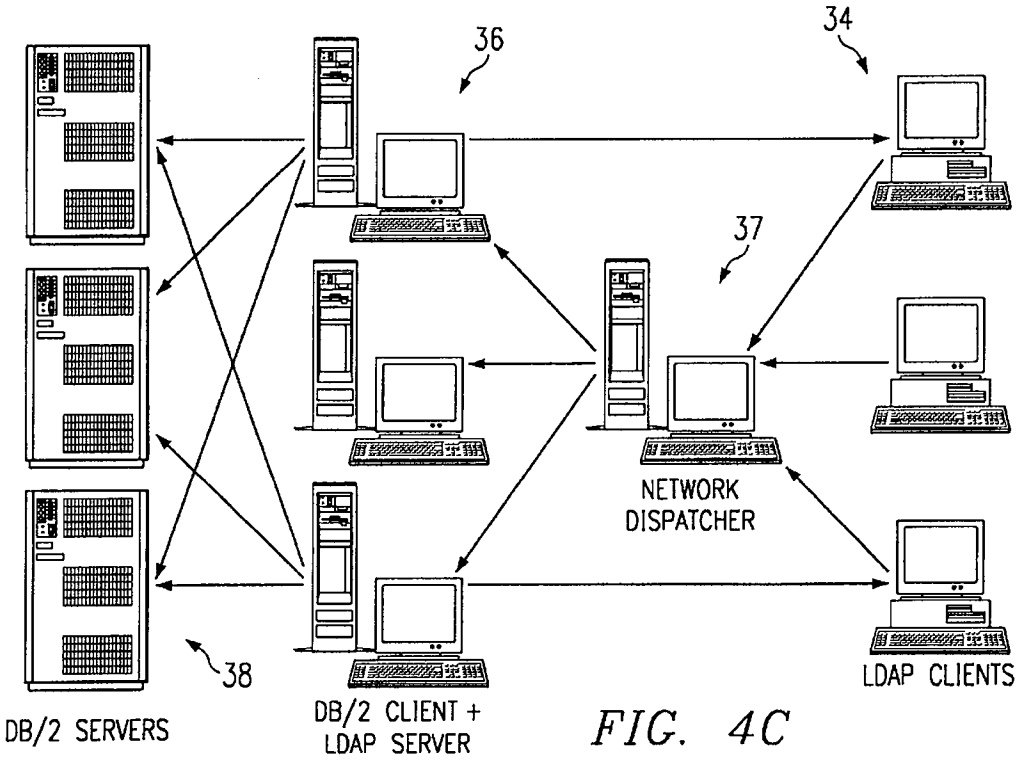

36
34
37
NETWORK
DISPATCHER
38
DB/2 SERVERS
DB/2 CLIENT +
LDAP SERVER
LDAP CLIENTS

FIG. 4C

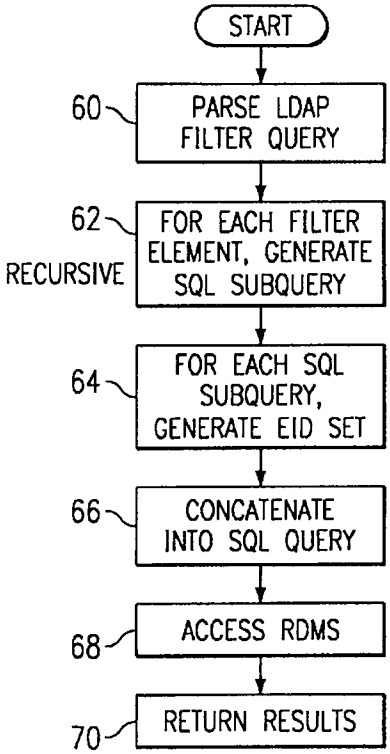
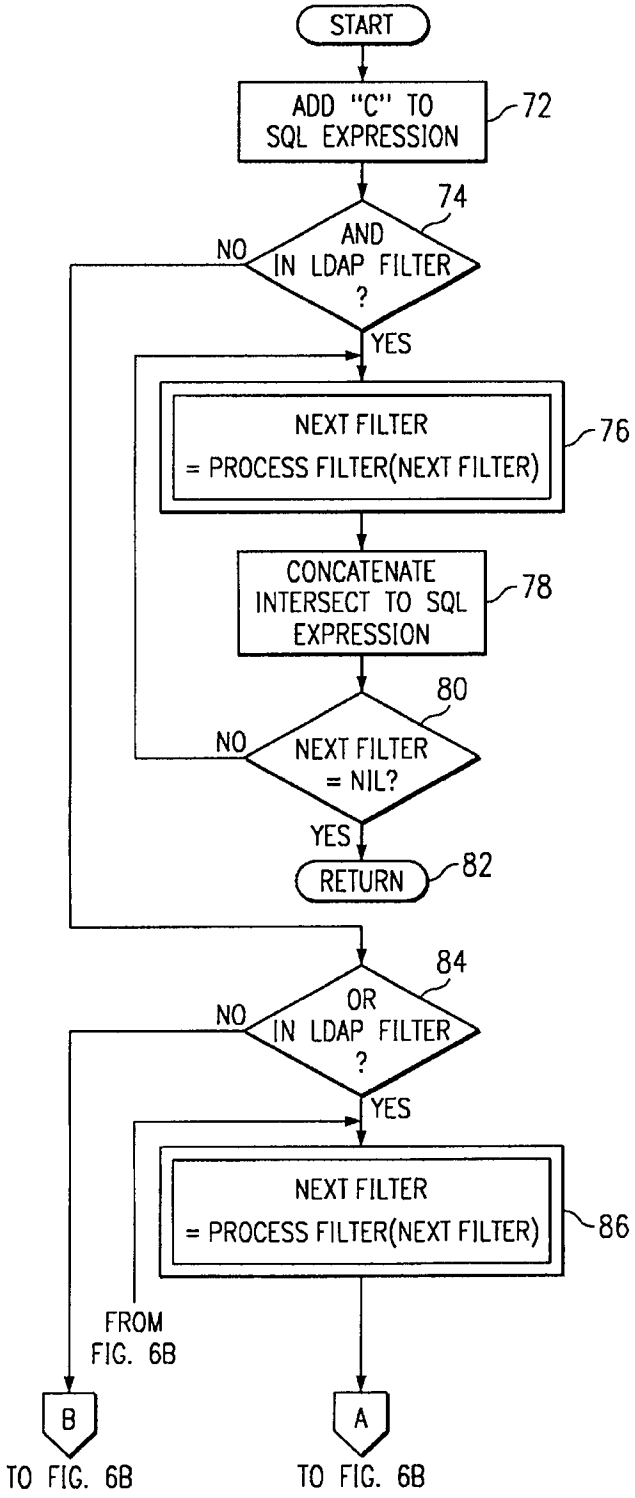

START
60
PARSE LDAP FILTER QUERY
62
RECURSIVE
FOR EACH FILTER ELEMENT, GENERATE SQL SUBQUERY
64
FOR EACH SQL SUBQUERY, GENERATE EID SET
66
CONCATENATE INTO SQL QUERY
68
ACCESS RDMS
70
RETURN RESULTS

FIG. 5

FIG. 6A


START
ADD "C" TO SQL EXPRESSION
72
74
AND IN LDAP FILTER ?
NO
YES
NEXT FILTER = PROCESS FILTER(NEXT FILTER)
76
CONCATENATE INTERSECT TO SQL EXPRESSION
78
80
NEXT FILTER = NIL?
NO
YES
RETURN
82
84
OR IN LDAP FILTER ?
NO
YES
NEXT FILTER = PROCESS FILTER(NEXT FILTER)
86
FROM FIG. 6B
B
TO FIG. 6B
A
TO FIG. 6B

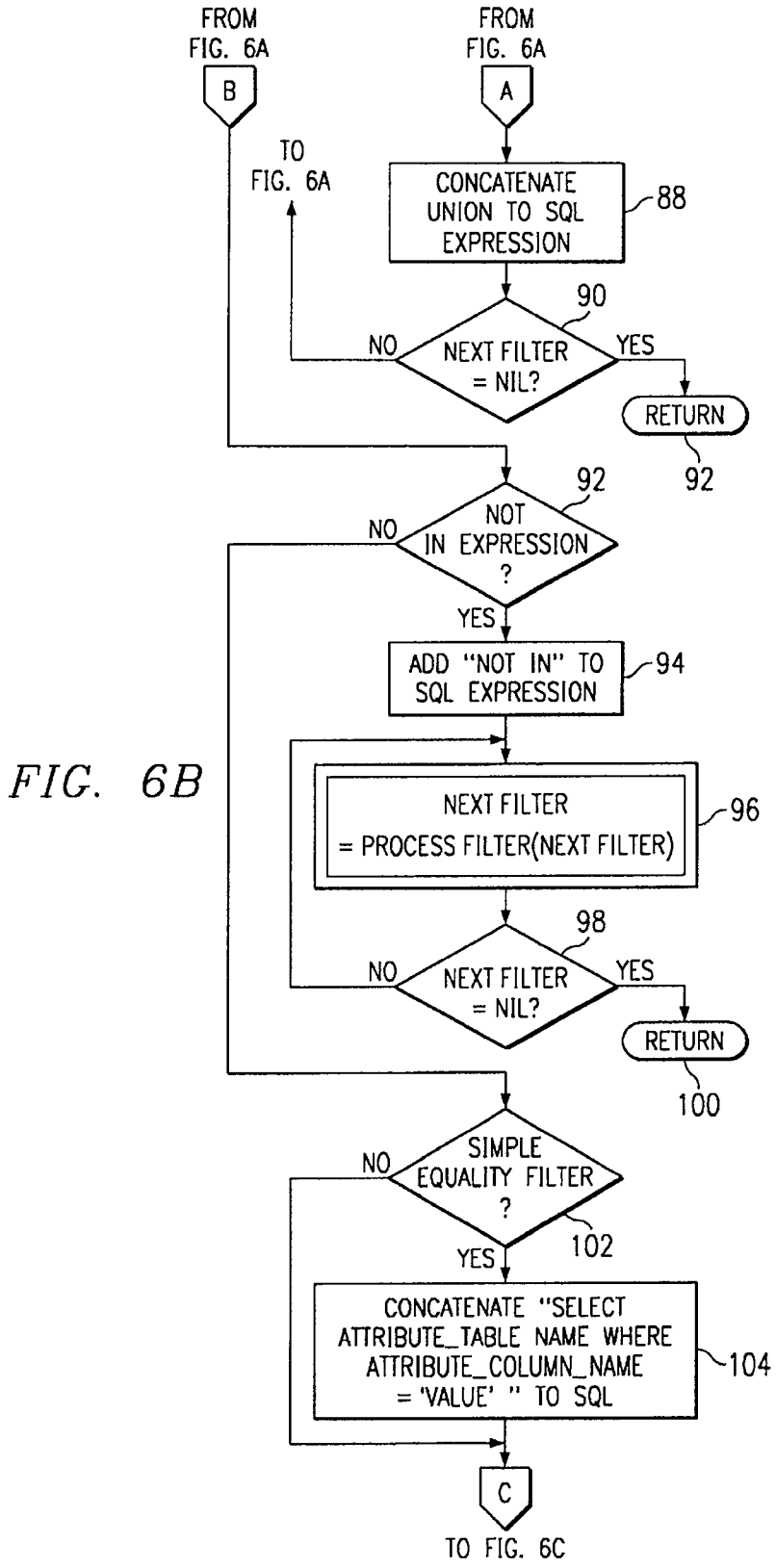

FROM FIG. 6A
B
FROM FIG. 6A
A
TO FIG. 6A
CONCATENATE UNION TO SQL EXPRESSION
88
90
NO
NEXT FILTER = NIL?
YES
RETURN
92
92
NO
NOT IN EXPRESSION ?
YES
ADD "NOT IN" TO SQL EXPRESSION
94
NEXT FILTER = PROCESS FILTER(NEXT FILTER)
96
98
NO
NEXT FILTER = NIL?
YES
RETURN
100
NO
SIMPLE EQUALITY FILTER ?
102
YES
CONCATENATE "SELECT ATTRIBUTE_TABLE NAME WHERE ATTRIBUTE_COLUMN_NAME = 'VALUE' " TO SQL
104
C
TO FIG. 6C

FIG. 6B

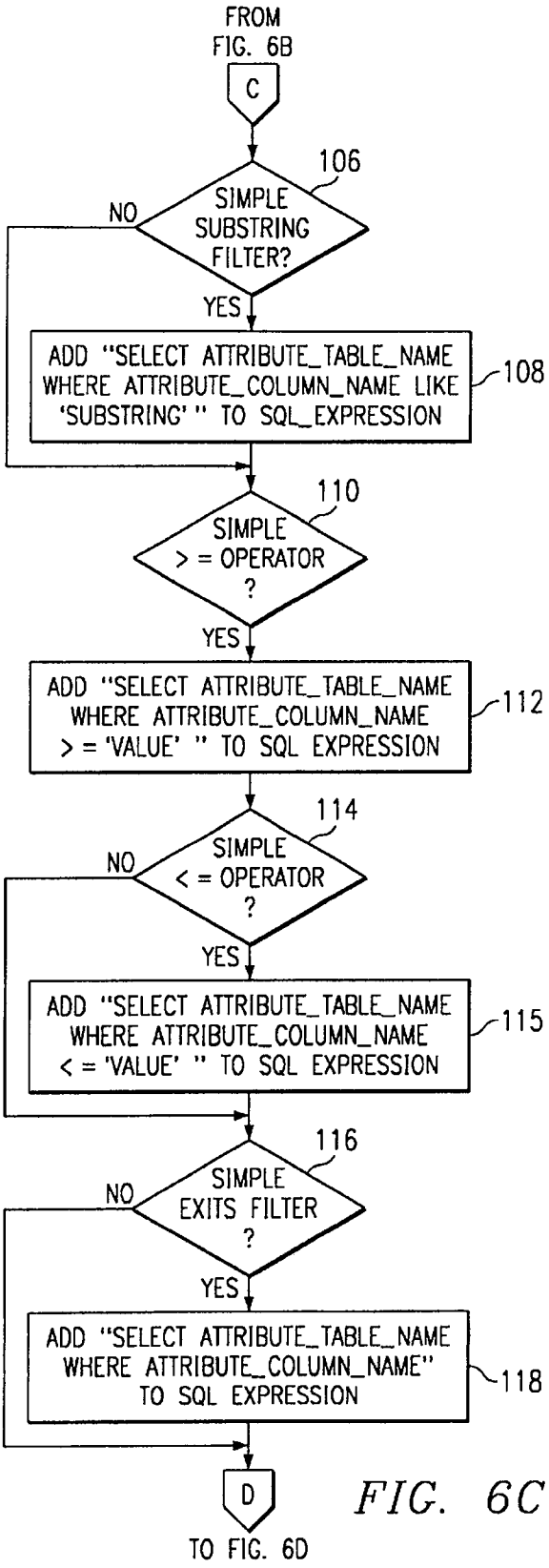

FROM FIG. 6B
C
106
SIMPLE SUBSTRING FILTER?
NO
YES
ADD "SELECT ATTRIBUTE_TABLE_NAME WHERE ATTRIBUTE_COLUMN_NAME LIKE 'SUBSTRING' " TO SQL_EXPRESSION
108
110
SIMPLE > = OPERATOR ?
YES
ADD "SELECT ATTRIBUTE_TABLE_NAME WHERE ATTRIBUTE_COLUMN_NAME > = 'VALUE' " TO SQL EXPRESSION
112
114
SIMPLE < = OPERATOR ?
NO
YES
ADD "SELECT ATTRIBUTE_TABLE_NAME WHERE ATTRIBUTE_COLUMN_NAME < = 'VALUE' " TO SQL EXPRESSION
115
116
SIMPLE EXITS FILTER ?
NO
YES
ADD "SELECT ATTRIBUTE_TABLE_NAME WHERE ATTRIBUTE_COLUMN_NAME" TO SQL EXPRESSION
118
D
TO FIG. 6D

FIG. 6C

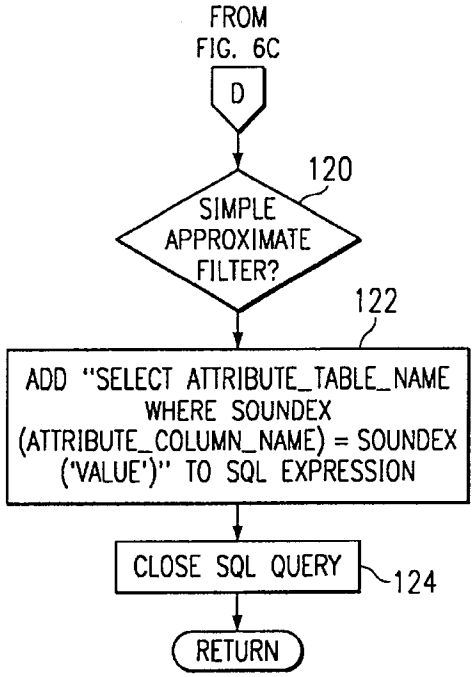

FROM FIG. 6C
D
120
SIMPLE APPROXIMATE FILTER?
122
ADD "SELECT ATTRIBUTE_TABLE_NAME WHERE SOUNDEX (ATTRIBUTE_COLUMN_NAME) = SOUNDEX ('VALUE')" TO SQL EXPRESSION
CLOSE SQL QUERY
124
RETURN

FIG. 6D

EFFICIENT IMPLEMENTATION OF LIGHTWEIGHT DIRECTORY ACCESS PROTOCOL (LDAP) SEARCH QUERIES WITH STRUCTURED QUERY LANGUAGE (SQL)

This application includes subject matter protected by copyright. All rights are reserved.

BACKGROUND OF THE INVENTION

1. Technical Field

This invention relates generally to providing directory services in a distributed computing environment.

2. Description of the Related Art

A directory service is the central point where network services, security services and applications can form an integrated distributed computing environment. Typical uses of a directory services may be classified into several categories. A "naming service" (e.g., DNS and DCE Cell Directory Service (CDS)) uses the directory as a source to locate an Internet host address or the location of a given server. A "user registry" (e.g., Novell NDS) stores information about users in a system composed of a number of interconnected machines. The central repository of user information enables a system administrator to administer the distributed system as a single system image. Still another directory service is a "white pages" lookup provided by some e-mail clients, e.g., Netscape Communicator, Lotus Notes, Endora and the like).

With more and more applications and system services demanding a central information repository, the next generation directory service will need to provide system administrators with a data repository that can significantly ease administrative burdens. In addition, the future directory service must also provide end users with a rich information data warehouse that allows them to access department or company employee data, as well as resource information, such as name and location of printers, copy machines, and other environment resources. In the Internet/intranet environment, it will be required to provide user access to such information in a secure manner.

To this end, the Lightweight Directory Access Protocol (LDAP) has emerged as an IETF open standard to provide directory services to applications ranging from e-mail systems to distributed system management tools. LDAP is an evolving protocol that is based on a client-server model in which a client makes a TCP/IP connection to an LDAP server, sends requests, and receives responses. The LDAP information model in particular is based on an "entry," which contains information about some object. Entries are typically organized in a specified tree structure, and each entry is composed of attributes.

LDAP provides a number of known functions including query (search and compare), update, authentication and others. The search and compare operations are used to retrieve information from the database. For the search function, the criteria of the search is specified in a search filter. The search filter typically is a Boolean expression that consists of qualifiers including attribute name, attribute value and Boolean operators like AND, OR and NOT. Users can use the filter to perform complex search operations. One filter syntax is defined in RFC 2254.

LDAP thus provides the capability for directory information to be efficiently queried or updated. It offers a rich set of searching capabilities with which users can put together

complex queries to get desired information from a backing store. Increasingly, it has become desirable to use a relational database for storing LDAP directory data. Representative database implementations include DB/2, Oracle, Sybase, Informix and the like. As is well known, Structured Query Language (SQL) is the standard language used to access such databases.

One of main goals for implementing an LDAP directory service with an relational database backing store (e.g., DB/2) is to provide a design and implementation such that all LDAP search queries can be executed efficiently with SQL. Implementing LDAP search queries with SQL, however, is a non-trivial task. On the one hand, because both LDAP and SQL use the same AND, OR and NOT logical operators, one possible solution to implementing LDAP search queries with SQL might simply involve mapping each LDAP operator to its corresponding SQL operator. This approach, however, does not work well in practice. Another approach, characterized by known prior art implementations (e.g., the Netscape b-tree LDAP server), involves retrieval of a superset of candidate entries, together with post-processing on those entries. There are several problems with this technique. The two-step process is time consuming. More problematic, however, is that negation and existence queries give rise to a sequential search through the whole database. For a LDAP directory with a large number of entries, search results cannot be returned in an efficient manner.

The present invention addresses the problem of efficiently mapping an LDAP filter into an SQL query.

BRIEF SUMMARY OF THE INVENTION

It is a primary object of this invention to provide a method for searching a relational database using hierarchical, filter-based queries, such as LDAP.

Another primary object is to provide an algorithm that combines basic LDAP filter expressions into a preferably single SQL query that retrieves target entries that exactly match given search criteria.

Still another important object of this invention is to provide a mechanism that can map even complicated LDAP queries having infinite logical depth into SQL to facilitate a relational database search.

Yet another important object of this invention is to map LDAP logical operators efficiently for use in an LDAP relational database search mechanism.

A more specific object of this invention is to efficiently implement LDAP search queries with SQL wherein simple queries are combined together to form an arbitrary complex query that can retrieve target entries, preferably with no post-processing involved.

It is also an object of the present invention to provide a method for mapping LDAP search queries into an SQL query that is efficient and does not degenerate into a sequential search.

It is another more specific object of the present invention to provide a recursive algorithm that can deal with LDAP filter operators in a consistent way, and that deals with complicated LDAP queries with infinite number of logical operators.

A more general object of this invention is to provide hierarchical LDAP searches using relational tables in an LDAP directory service having a relational database management system (DBMS) as a backing store.

A more general object of this invention is to provide a reliable and scaleable enterprise directory solution, wherein a preferred implementation is LDAP using a DB/2 backing store.

The present invention solves the problem of efficiently mapping an LDAP filter into an SQL query using unique entry identifier (EID) sets. According to the inventive method, a SQL subquery is first generated for each LDAP operator based on given translation rules. The SQL subquery generates a set of entry EIDs that match the LDAP basic operation. Thereafter, the SQL subqueries are combined into a single SQL query according to a set of combination rules chosen corresponding to the logical operators of the LDAP filter query. Thus, for example, if the LDAP logical operator is OR(|), the invention then preferably uses an SQL UNION to union the sets generated from the subquery. If the LDAP logical operator is AND (&), the invention preferably uses an SQL INTERCEPT to intercept the sets generated from the subquery. If the LDAP logical operator is NOT, the invention preferably excludes entries by negating the IN operation before the subquery. Thus, the combination rules includes, for example, mapping the LDAP logical OR operation to an SQL UNION, mapping the LDAP logical operation AND to SQL INTERCEPT, and mapping the LDAP logical operation NOT to SQL NOT IN.

Generalizing, according to the preferred embodiment, a method for searching a relational database using hierarchical, filter-based queries begins by parsing a filter-based query for elements and logical operators of the filter query. For each filter element, the method generates an SQL subquery according to a set of translation rules. For each SQL subquery, the method then generates a set of entry identifiers for the filter query. Then, the SQL subqueries are combined into a single SQL query according to a set of combination rules chosen corresponding to the logical operators of the filter query.

The foregoing has outlined some of the more pertinent objects of the present invention. These objects should be construed to be merely illustrative of some of the more prominent features and applications of the invention. Many other beneficial results can be attained by applying the disclosed invention in a different manner or modifying the invention as will be described. Accordingly, other objects and a fuller understanding of the invention may be had by referring to the following Detailed Description of the preferred embodiment.

BRIEF DESCRIPTION OF THE DRAWINGS

For a more complete understanding of the present invention and the advantages thereof, reference should be made to the following Detailed Description taken in connection with the accompanying drawings in which:

FIG. **1** is a representative LDAP directory service implementation;

FIG. **2** is a simplified LDAP directory;

FIG. **3** is a flowchart of an LDAP directory session;

FIGS. **4A–4C** show representative LDAP directory service implementations having relational database backing store;

FIG. **5** is a simplified flowchart of the inventive method for hierarchical LDAP searching in an LDAP directory service having a relational database management system as a backing store using LDAP filter queries mapped efficiently to SQL; and

FIGS. **6A–6D** are a detailed composite flowchart of a preferred embodiment of the inventive recursive algorithm of the invention that is used to generate the SQL query.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

A block diagram of a representative LDAP directory service in which the present invention may be implemented is shown in FIG. **1**. As is well-known, LDAP is the light-weight directory access protocol, and this protocol has been implemented in the prior art, e.g., as either a front end to the X.500 directory service, or as a standalone directory service. According to the protocol, a client machine **10** makes a TCP/IP connection to an LDAP server **12**, sends requests and receives responses. LDAP server **12** supports a directory **21** as illustrated in a simplified form in FIG. **2**. Each of the client and server machines further include a directory "runtime" component **25** for implementing the directory service operations as will be described below. The directory **21** is based on the concept of an "entry" **27**, which contains information about some object (e.g., a person). Entries are composed of attributes **29**, which have a type and one or more values. Each attribute **29** has a particular syntax that determines what kinds of values are allowed in the attribute (e.g., ASCII characters, jpeg file, etc.) and how these values are constrained during a particular directory operation.

The directory tree is organized in a predetermined manner, with each entry uniquely named relative to its sibling entries by a "relative distinguished name" (RDN). An RDN comprises at least one distinguished attribute value from the entry and, at most, one value from each attribute is used in the RDN. According to the protocol, a globally unique name for an entry, referred to as a "distinguished name" (DN), comprises a concatenation of the RDN sequence from a given entry to the tree root.

The LDAP search can be applied to a single entry (a base level search), an entry's children (a one level search), or an entire subtree (a subtree search). Thus, the "scope" supported by LDAP search are: base, one level and subtree. LDAP does not support search for arbitrary tree levels and path enumeration.

LDAP includes an application programming interface (API), as described in "The C LDAP Application Program Interface", IETF Working Draft, Jul. 29, 1997, which is incorporated herein by reference. An application on a given client machine uses the LDAP API to effect a directory service "session" according to the flowchart of FIG. **3**. At step **40**, an LDAP session with a default LDAP server is initialized. At step **42**, an API function ldap_init( ) returns a handle to the client, and this handle may allow multiple connections to be open at one time. At step **44**, the client authenticates to the LDAP server using, for example, an API ldap_bind( ) function. At step **46**, one or more LDAP operations are performed. For example, the API function ldap_search( ) may be used to perform a given directory search. At step **48**, the LDAP server returns the results of the directory search, e.g., one or more database elements that meet the search criteria. The session is then closed at step **50** with the API ldap_unbind( ) function then being used to close the connection.

It may be desirable to store LDAP directory data in a backing store. FIGS. **4A–4C** illustrates several representative LDAP directory service implementations that use a relational database management system (RDBMS) for this purpose. These systems merely illustrate possible LDAP directory services in which the present invention may be implemented. One of ordinary skill should appreciate, however, that the invention is not limited to an LDAP directory service provided with a DB/2 backing store. The principles of the present invention may be practiced in other types of directory services (e.g., X.500) and using other relational database management systems (e.g., Oracle, Sybase, Informix, and the like) as the backing store.

In FIG. **4A**, an LDAP client **34** can connect to a number of networked databases **38***a*–**58***n* through an LDAP server

36. The databases 38*a*–38*n* contain the directory information. However, from the user's perspective, the LDAP server 36 stores all the information without knowing the database 38 in which the data is actually located. With this configuration, the LDAP server 36 is freed from managing the physical data storage and is able to retrieve information from multiple database servers 38 which work together to form a huge data storage.

FIG. 4B illustrates a multiple client/multiple server LDAP/DB2 enterprise solution. In this environment, a DB/2 client preferably runs on each LDAP server 36. Each such DB/2 client can connect to any database server 38 containing directory information. The collection of database servers 38*a*–38*n* form a single directory system image, and one or more of the LDAP servers 36 can access such information. Because all the LDAP servers 36 see the same directory image, a network dispatcher 37 may be deployed to route requests among the LDAP servers 36.

FIG. 4C illustrates a multiple client/parallel super server configuration. In certain environments, where users need to store large amounts of information into the directory, this configuration automatically partitions the database into different machines 38. In addition, database queries are divided into smaller, independent tasks that can execute concurrently, which increases end user query response time. This configuration enables users to store up to terabytes of data into the database.

One of ordinary skill should appreciate that the system architectures illustrated in FIGS. 4A–4C are not to be taken as limiting the present invention. The inventive technique may be used to search any relational database using hierarchical, filter-based database queries.

The technique is now described generally with reference to the flowchart of FIG. 5. This diagram illustrates the basic method of the invention to provide hierarchical LDAP searching in an LDAP directory service having a relational database management system (DBMS) as a backing store. The method begins at step 60 by parsing an LDAP filter-based query for elements and logical operators of the filter query. For each filter element, the method continues at step 62 to generate an SQL subquery according to a set of translation rules, which will be defined below. For each SQL subquery, the method then continues at step 64 to generate a set of unique entry identifiers for the LDAP filter query. Then, at step 66, the SQL subqueries are combined into a single SQL query according to a set of combination rules chosen corresponding to the logical operators of the LDAP filter query. At step 68, the single SQL query is applied to the database step. Results are returned at step 70.

The specific details of the routine illustrated in FIG. 5 are now described. By way of brief background, the inventive scheme preferably takes advantage of several LDAP table structures that are now described below. Further details about these structures are provided in U.S. Ser. No. 09/050,503 titled "A Fast And Efficient Method To Support Hierarchical LDAP Searches With Relational Tables", assigned to the assignee of this application, and incorporated herein by reference.

Entry Table

This table holds the information about a LDAP entry. This table is used for obtaining the EID of the entry and supporting LDAP_SCOPE_ONELEVEL and LDAP_SCOPE_BASE search scope.

EID. The unique identifier of the LDAP entry. This field is indexed.

PEID. The unique identifier of a parent LDAP entry in the naming hierarchy.

EntryData. Entries are stored using a simple text format of the form "attribute: value". Non-ASCII values or values that are too long to fit on a reasonable sized line are represented using a base 64 encoding. Giving an ID, the corresponding entry can be returned with a single SELECT statement.

Descendant Table

The purpose of this table is to support the subtree search feature of LDAP. For each LDAP entry with an unique ID (AEID), this table contains the descendant entries unique identifiers (DEID). The columns in this table are:

AEID. The unique identifier of the ancestor LDAP entry. This entry is indexed.

DEID. The unique identifier of the descend LDAP entry. This entry is indexed.

Attribute Table

One table per searchable attribute. Each LDAP entry is assigned an unique identifier (EID) by the backing store. The columns for this table are:

EID

Attribute value

Thus, in the parent table, the EID field is the unique identifier of an entry in the LDAP naming hierarchy. The PEID field is the unique identifier of the parent entry in the naming hierarchy. In the descendant table, the AEID field is the unique identifier of a ancestor LDAP entry in the LDAP naming hierarchy. The DEID field is the unique identifier of the descend LDAP entry.

In addition to the table structures described above, the following SQL SELECT statements are used by LDAP/DB2 search routines:

Base Level Search:

```
SELECT entry.EntryData,
from ldap_entry as entry
where entry.EID in (
        select distinct ldap_entry.EID
        from <table list>
        where (ldap_entry.EID=<root dn id> )
            <sql where expressions>)
```

One Level Search:

```
SELECT entry.EntryData,
   from ldap_entry as entry
   where distinct ldap_entry.EID
   from ldap_entry, <table list>
      ldap_entry as pchild, <list of tables>
      where ldap_entry.EID=pchild.EID
      AND pchild.PIED=<root dn id>
          <sql where expressions>)
```

Subtree Search

```
SELECT entry.EntryData,
   from ldap_entry as entry
   where entry.EID in (
      select distinct ldap_entry.EID
      from ldap_entry, ldap_desc, <table list>
      where
      (LDAP_ENTRY.EID=ldap_desc.DEID AND
      ldap_desc.AEID=<root dn id>)
      ldap_entry as pchild. <table list>
      where ldap_entry.EID=ldap_desc.EID
      AND ldap_desc.AEID=%d <where expressions>).
```

In the above representation, <table list> and <where expression> are the two null terminated strings returned by the SQL generator. The <root dn id> is the unique identifier of the root dn. The where clause should only be generated if <where expression> is not the empty string and no errors where detected in the parsing the LDAP filter.

As is well-known, LDAP search queries comprise six basic filters with the format <attribute> <operator> <value>. Complex search filters are generated by combining basic filters with Boolean operators AND (&), OR (|) and NOT (!).

Translation Rules

As described above in the flowchart of FIG. **5**, an SQL subquery is generated for each LDAP filter element according to a set of translation rules. This was step **62** in the method. The following sets forth preferred translation rules used to generate SQL queries for basic LDAP filters according to the present invention:

Equality

LDAP filter

(<attr> = <value>)

SQL Expression

WHERE columnname = 'value'

Ranges:

LDAP filter

(<attr> = <value>)

(<attr> = <value>)

SQL Expression

WHERE columnname = 'value'

WHERE columnname = 'value'

Substring:

LDAP filter

(<attr> = ·<leading>"*·any"*·<trailing>")

SQL Expression

WHERE columnname

LIKE ·<leading>"*·any"*·<trailing>"

Approximate:

LDAP filter

(<attr> = <value>)

SQL Expression

WHERE SOUNDEX (columnname) = SOUNDEX ('value')

As described above, according to the inventive method, for each LDAP filter element or sub-expression, there is a set of entries (EIDs) that will satisfy the element. Thus, each element generally maps to a set of EIDs. The EID sets are then merged together, preferably into a single SQL query, using a set of combination rules. Thus, if a pair of LDAP filter elements are subject to an LDAP logical OR operator, the corresponding EID sets are merged using an SQL UNION logical operator. If a pair of LDAP filter elements are subject to an LDAP logical AND operator, the corresponding EID sets are merged using an SQL INTERSECT logical operator. If a pair of LDAP filter elements are subject to an LDAP logical NOT operator, the corresponding EID sets are merged using an SQL NOT IN logical operator. As will also be seen, these combination rules are applied recursively such that all LDAP elements associated with a particular logical operator are processed into the SQL query. This recursive processing facilitates handling of even complicated LDAP queries having numerous layers of logical depth.

With the basic translation rules and the EID sets approach, the present invention thus uses a recursive algorithm that handles complicated queries with a large number of layers of logical operators. The SQL generation algorithm of the present invention is illustrated in the composite flowchart of FIGS. 6A–6D. Note that the algorithm presented may be applied to search queries for all levels.

With reference to FIG. **6A**, the method begins at step **72** by concatenating "(" to the generated SQL query (which, in the first iteration, is otherwise a skeleton query). The routine then tests at step **74** to determine whether the LDAP filter element includes the AND logical operator. If the outcome of the test at step **74** is positive, the AND logical operator is present. Typically, an AND logical operator sets apart at least a pair of subexpressions, and thus the routine includes appropriate logic to handle each subexpression separately in a recursive manner. To this end, the routine continues at step **76** and sets an process variable nextFilter equal to ProcessFilter(nextFilter). At step **78**, the routine concatenates the SQL INTERSECT operator to the SQL query. This step maps the LDAP AND logical operator in the manner previously described. At step **80**, a test is made to determine whether nextFilter is nil, i.e. whether all subexpressions associated with the AND logical operator in the LDAP filter have been processed. If not, the routine returns at step **82**. If, however, the outcome of the test at step **80** is negative, the routine loops back to step **76** to process the next subexpression associated with the AND logical operator currently being mapped.

If the outcome of the test at step **74** is negative, which indicates that the AND logical element is not in the filter element, the routine branches to step **84** to determine whether the LDAP OR logical operator is present. If the outcome of the test at step **84** is negative, the routine branches to FIG. **6B**. If, however, the outcome of the test at step **84** is positive, the routine continues at step **86** with a recursive call into the ProcessFilter(nextFilter) subroutine previously described. Just as with the AND logical operator, an OR logical operator will typically include subexpressions that must be processed in a recursive manner.

With reference now to FIG. **6B**, the routine then continues at step **88** to concatenate the SQL UNION operator into the SQL query. At step **90**, a test is performed to determine whether there are any more subexpressions to test. If not, the routine returns at step **92**. If, however, the outcome of the test at step **90** is negative, the routine loops back to step **86** to process the next subexpression associated with the OR logical operator currently being mapped.

If the LDAP filter element includes neither AND nor OR, the routine continues at step **92** to determine whether the NOT logical operator is present. If so, the routine continues at step **94** to add the NOT IN logical operator to the SQL expression being generated. The routine then continues at step **96** to enter the recursive call so that all associated subexpressions may be parsed through the algorithm in the manner previously described. Thus, at step **98**, a test is performed to determine whether all subexpressions associated with the NOT operator have been processed. If so, the routine returns at step **100**; otherwise, the routine loops back to step **96** and processes the next subexpression.

The remainder of the flowchart describes the mapping of simple filter elements using the set of translation rules previously described. At step **102**, a test is made to determine whether the LDAP filter includes a simple equality statement (e.g., "a=1"). If so, the routine continues at step **104** to concatenate the associated attribute column name value into the SQL query. Thereafter, or if the result of the test at step **102** is negative, the routine continues in FIG. **6C**.

At step **106**, a test is made to determine whether the LDAP filter element is a substring filter. If so, the routine continues at step **108** to concatenate the associated attribute column name value into the SQL query. Thereafter, or if the result of the test at step **106** is negative, the routine continues at step **110** to determine whether the LDAP filter element is a greater than or equal expression. If so, the routine continues at step **112** to concatenate the associated attribute

column name substring value into the SQL query being constructed. Thereafter, the routine continues at step **114** to determine whether the LDAP filter element is a less than or equal to expression. If so, the routine concatenates the associated value into the SQL expression at step **115** and continues at step **116**. Step **116** is reach also as a result of a negative outcome of step **114**.

At step **116**, a test is performed to determine whether the LDAP filter expression is a simple exists filter. If so, the routine concatenates the associated value into the SQL expression at step **118**. The routine then continues with step **120** in FIG. **6**D, which is also reached by a negative outcome to the test at step **116**. Step **120** tests whether the LDAP filter includes the simple approximate filter. If so, the appropriate value is concatenated into the SQL query at step **122**. At step **124**, the SQL query is closed by concatenating an ")" value to complete the processing.

The following is a detailed listing of a preferred code implementation of the above-described recursive algorithm:

```
BEGIN PROCESSFILTER ALGORITHM ON AN LDAP-FILTER.
  CONCATENATE "(" TO SQL-EXPRESSION.
  DO ONE OF THE FOLLOWING DEPENDING ON THE TYPE OF
  LDAP FILTER:
  1) For complex ldap_filter with an AND operation:
SET NEXTFILTER = LDAP-FILTER1
LOOP UNTIL NEXTFILTER IS EMPTY
  SET NEXTFILTER - PROCESSFILTER (NEXTFILTER)
  CONCATENATE" INTERSECT" TO SQL-EXPRESSION.
END OF LOOP
  2) For complex ldap-filter with an OR operation:
SET NEXTFILTER - LDAP-FILTER1
LOOP UNTIL NEXTFILTER IS EMPTY
  SET NEXTFILTER - PROCESSFILTER (NEXTFILTER)
  CONCATENATE" UNION" TO SQL-EXPRESSION.
END OF LOOP
  3) For complex ldap-filter with a NOT operation:
ADD " NOT " TO SQL-EXPRESSION.
  PROCESSFILTER (LDAP-FILTER1)
  4) For simple equality filter:
CONCATENATE "SELECT ATTRIBUTE_TABLE_NAME WHERE
  ATTRIBUTE_COLUMN_NAME = 'VALUE' "
  TO SQL-EXPRESSION.
  5) For simple substring filter:
CONCATENATE "SELECT ATTRIBUTE_TABLE_NAME WHERE
  ATTRIBUTE_COLUMN_NAME LIKE' SUB-STRING' "
  TO SQL-EXPRESSION.
  6) For simple greater or equal filter:
CONCATENATE "SELECT ATTRIBUTE_TABLE_NAME WHERE
  ATTRIBUTE_COLUMN_NAME>= 'VALUE' "
  TO SQL-EXPRESSION.
  7) For simple less or equal filter:
CONCATENATE "SELECT ATTRIBUTE_TABLE_NAME WHERE
  ATTRIBUTE_COLUMN_NAME <= 'VALUE' "
  TO SQL-EXPRESSION.
  8) For simple attribute exists filter:
CONCATENATE "SELECT ATTRIBUTE_TABLE_NAME"
  TO SQL-EXPRESSION.
  9) For simple approximate filter:
CONCATENATE "SELECT ATTRIBUTE_TABLE_NAME WHERE
  SOUNDEX (ATTRIBUTE_COLUMN_NAME) = SOUNDEX ('VALUE')"
  TO SQL-EXPRESSION.
  Concatenate ")" to SQL-expression.
  Return next - ldap-filter.
End processFilter algorithm.
```

EXAMPLES

With the basic translation rules and the EID sets approach described in the flowcharts of FIG. **5** and FIGS. **6**A–**6**D, the following are the SQL queries that the invention generates for a representative LDAP filter query of the following form (|(f1='v1') (f2='v2')):

```
Alternative 1
  SELECT entry.EntryData
  FROM LDAP_ENTRY as entry
  WHERE entry.EID in
  (
  SELECT distinct LDAP_ENTRY.EID
  FROM ldap_entry, ldap_desc, f1
  WHERE
    (ldap_entry.EID=ldap_desc.DEID AND
    ldap_desc_AID=<id>) AND
    ldap_entry.eid=f1.eid AND
    f1='v1')
    UNION
    SELECT distinct ldap_entry.EID
    FROM ldap_entry, ldap_desc, f2
    WHERE (ldap_entry.EID=ldap+desc.DEID ADND
    ldap_desc.AEID=<id>)
    AND ldap_entry.EID=f2.eid
    AND f2='v2'))
Alternative 2
  SELECT entry.EntryData
  FROM LDAP_ENTRY as entry WHERE entry.EID in
  ( SELECT distinct LDAP_ENTRY.EID FROM
    LDAP_ENTRY,ldap_desc
WHERE
  (LDAP_ENTRY.EID=ldap_desc.DEID AND ldap_
    desc.AEID=<id>)
  AND LDAP_ENTRY.EID
  IN ((SELECT EID FROM f1 WHERE f1 = ' v1')
  UNION (SELECT EID FROM SN WHERE SN =' v2' )))
```

Both SQL statements illustrated above generate the correct search results, and both techniques are within the scope of the present invention. As can be seen, the first query performs the JOIN operation with the ldap descendant table within each subquery. The second query performs the JOIN operation with the ldap descendant table outside the subquery. Although either alternative may be used to implement the present invention, Alternative 2 may provide better performance results. In addition to correct results, the OR operation (illustrated above) performs well with both alternative techniques because irrelevant entries are filtered out in the subquery and target entries are reported back to the main query.

As noted above, with the set-based approach of the present invention, the LDAP NOT operation preferably is performed by excluding entries through negating the IN operation before the subquery. The following example illustrates the operation:

Filter String:

(!(f1='v1'))

SQL Statement:

```
SELECT entry.EntryData,
FROM LDAP_ENTRY as entry
WHERE entry.EID in
( SELECT distinct LDAP_ENTRY.EID FROM
  LDAP_ENTRY.ldap_desc
WHERE (LDAP_ENTRY.EID=ldap_desc.DEID AND
  ldap_desc.AEID=<id>)
AND LDAP_ENTRY.EID NOT IN ((SELECT EID
  FROM f1
  where f1=' v1')))
```

The following is another example of a SQL statement generated for complex query with AND, OR and NOT operator.

Complex Query with AND, OR and NOT Operator

Filter String:

(&(| (objectclass=PERSON) (objectclass=GROUP)) (sn= SMITH) (!(member=*)))

SQL Statement:

```
SELECT entry.EntryData,
FROM LDAP_ENTRY as entry WHERE entry.EID in
( SELECT distinct LDAP_ENTRY.EID FROM
  LDAP_ENTRY,ldap_desc
WHERE (LDAP_ENTRY.EID=ldap_desc.DEID AND
  ldap_desc.AEID=?) AND
LDAP_ENTRY.EID
IN (((SELECT EID FROM OBJECTCLASS WHERE
  OBJECTCLASS =
PERSON)
UNION (SELECT EID FROM OBJECTCLASS
  WHERE OBJECTCLASS =
GROUP))
INTERSECT (SELECT EID FROM SN WHERE SN =
  SMITH )
  INTERSECT
(SELECT EID FROM LDAP_ENTRY WHERE EID
  NOT IN
(SELECT EID FROM MEMBER))))
```

It should be noted that the subtree search level is illustrated in the examples herein, but only for purposes of illustration. This should not be taken by way of limitation.

As noted above, the invention may be implemented in any hierarchical directory service in which a relational database management system (RDBMS) is used to provide a backing store function. Thus, for example, the principles of the invention may be carried out in an X.500 directory service or hereinafter-developed LDAP implementations. The SQL query generated according to the present invention is used to access the relational database, and results are then returned in response to this query. The invention may also be implemented within a relational database management system being used as an add-on to a directory service. One of ordinary skill will appreciate that the invention can be applied to any relational database management system (RDBMS) and not simply DB/2, the implementation described above. Thus, for example, the relational database may be Oracle, Sybase or any other third party supplied backing store. In addition, the EID sets approach can also be applied to b-tree based LDAP server implementation.

Moreover, although the preferred embodiment has been described in the context of generating a Structured Query Language (SQL) query, the inventive technique should be broadly construed to extend to any relational database query language.

One of the preferred embodiments of the routines of this invention is as a set of instructions (computer program code) in a code module resident in or downloadable to the random access memory of a computer.

Having thus described our invention, what we claim as new and desire to secure by Letters Patent is set forth in the following claims.

What is claimed is:

1. A method for searching a relational database using hierarchical, filter-based queries, comprising the steps of:

parsing a filter-based query for elements and logical operators of the filter query;

for each filter element, generating an SQL subquery according to a set of translation rules;

for each SQL subquery, generating a set of entry ID's for the filter query; and

combining the SQL subqueries into a single SQL query according to a set of combination rules chosen corresponding to the logical operators of the filter query.

2. The method as described in claim **1** wherein the filter-based query is a Lightweight Directory Access Protocol (LDAP) directory service query.

3. The method as described in claim **2** wherein the logical operators of the LDAP filter-based query include AND, OR and NOT.

4. The method as described in claim **3** wherein the combination rules map the OR logical operator to an SQL UNION operator.

5. The method as described in claim **3** wherein the combination rules map the AND logical operator to an SQL INTERCEPT operator.

6. The method as described in claim **3** wherein the combination rules map the NOT logical operator to an SQL NOT IN operator.

7. The method as described in claim **2** wherein the relational database is DB/2.

8. The method as described in claim **1** further including the step of accessing the relational database using the single SQL query.

9. The method as described in claim **8** further including the step of returning a response to the single SQL query.

10. The method as described in claim **1** wherein the step of combining the SQL subqueries is carried out recursively until all filter elements of the filter query have been processed into the SQL query.

11. A method for searching a relational database from a Lightweight Directory Access Protocol (LDAP) directory service generating filter-based queries, comprising the steps of:

parsing an LDAP filter-based query for elements and logical operators of the LDAP filter query;

for each LDAP filter element, generating an SQL subquery according to a set of translation rules;

for each SQL subquery, generating a set of entry ID's for the LDAP filter query; and

combining the SQL subqueries into a single SQL query according to a set of combination rules chosen corresponding to the logical operators of the LDAP filter query.

12. The method as described in claim **11** wherein the logical operators of the LDAP filter-based query include AND, OR and NOT.

13. The method as described in claim **12** wherein the combination rules map the OR logical operator to an SQL UNION operator.

14. The method as described in claim **12** wherein the combination rules map the AND logical operator to an SQL INTERCEPT operator.

15. The method as described in claim **12** wherein the combination rules map the NOT logical operator to an SQL NOT IN operator.

16. The method as described in claim **11** wherein the step of combining the SQL subqueries is carried out recursively until all filter elements of the filter query have been processed into the SQL query.

17. A computer program product in computer-readable media for searching a relational database using hierarchical, filter-based queries, comprising:

means for parsing a filter-based query for elements and logical operators of the filter query;

means for generating an SQL subquery for each filter element according to a set of translation rules;

means for generating a set of entry ID's for each SQL subquery; and

means for combining the SQL subqueries into a single SQL query according to a set of combination rules chosen corresponding to the logical operators of the filter query.

18. The computer program product as described in claim **17** wherein the filter-based query is a Lightweight Directory Access Protocol (LDAP) directory service query.

19. The computer program product as described in claim **18** wherein the logical operators of the LDAP filter-based query include AND, OR and NOT.

20. The computer program product as described in claim **19** wherein the combination rules map the OR logical operator to an SQL UNION operator.

21. The computer program product as described in claim **19** wherein the combination rules map the AND logical operator to an SQL INTERCEPT operator.

22. The computer program product as described in claim **19** wherein the combination rules map the NOT logical operator to an SQL NOT IN operator.

23. The computer program product as described in claim **18** wherein the relational database is DB/2.

24. A directory service, comprising:

a directory organized as a naming hierarchy having a plurality of entries each represented by a unique identifier;

a relational database management system having a backing store for storing directory data;

means for searching the directory, comprising:

means for parsing an hierarchical, filter-based query for elements and logical operators of the filter query;

means for generating a relational database subquery for each filter element according to a set of translation rules;

means for generating a set of unique identifiers for each relational database subquery; and

means for combining the relational database subqueries into a single relational database query according to a set of combination rules chosen corresponding to the logical operators of the filter query.

25. The directory service as described in claim **24** wherein the directory is compliant with the Lightweight Directory Access Protocol (LDAP).

26. The directory service as described in claim **25** wherein the relational database management system is DB/2.

27. In a directory service having a directory organized as a naming hierarchy, the hierarchy including a plurality of entries each represented by a unique identifier, the improvement comprising:

a relational database management system having a backing store for storing directory data;

means for searching the directory, comprising:

means for parsing an hierarchical, filter-based query for elements and logical operators of the filter query;

means for generating a relational database subquery for each filter element according to a set of translation rules;

means for generating a set of unique identifiers for each relational database subquery; and

means for combining the relational database subqueries into a single relational database query according to a set of combination rules chosen corresponding to the logical operators of the filter query.

28. In the directory service as described in claim **27** wherein the directory is compliant with the Lightweight Directory Access Protocol (LDAP).

29. In the directory service as described in claim **28** wherein the relational database management system is DB/2.

* * * * *

US005412804A

United States Patent [19]

Krishna

[11] Patent Number: 5,412,804

[45] Date of Patent: May 2, 1995

[54] **EXTENDING THE SEMANTICS OF THE OUTER JOIN OPERATOR FOR UN-NESTING QUERIES TO A DATA BASE**

[75] Inventor: **Murali M. Krishna,** Colorado Springs, Colo.

[73] Assignee: **Oracle Corporation,** Redwood City, Calif.

[21] Appl. No.: **876,393**

[22] Filed: **Apr. 30, 1992**

[51] **Int. Cl.**[6] **G06F 15/40**

[52] **U.S. Cl.** **395/600;** 364/DIG. 1; 364/282.1; 364/283.4

[58] **Field of Search** 395/600, 700, 425

[56] **References Cited**

U.S. PATENT DOCUMENTS

4,506,326	3/1985	Shaw et al.	395/700
4,648,044	3/1987	Hardy et al.	364/513
4,829,427	5/1989	Green	395/600
4,918,593	4/1990	Huber	395/600
4,956,774	9/1990	Shibamiya et al.	395/600
5,276,870	1/1994	Shan et al.	395/600

OTHER PUBLICATIONS

R. Epstein, "Techniques for Processing of Aggregates in Relational Database Systems," Memorandum No. UCB/ERL M79/8, Electronics Research Lab., UCLA, Berkeley, Calif. 21 Feb. 1979.

Hobbs and England, *Rdb/VMS—A Comprehensive Guide,* Digital Equipment Corporation, Maynard, Mass. (1991).

Arnon Rosenthal and Cesar Galindo-Legaria, "Query Graphs, Implementing Trees, and Freely-Reorderable Outerjoins," Proc. SIGMOD Conf., Association for Computing Machinery, United States (May 1990), pp. 291–299.

M. Muralikrishna, "Optimization and Dataflow Algorithms for Nested Tree Queries," Proc. VLDB Conf. (Aug. 1989), pp. 77–85.

Ganski & Wong, "Optimization of Nested SQL Queries Revisited," Proc. SIGMOD Conf., Association for Computing Machinery, United States (May 1987), pp. 22–23.

Umeshwar Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates and Quantifiers," Proceedings of the 13 VLDB Conference, Brighton, 1987, pp. 197–208.

C. J. Date, *A Guide to The SQL Standard,* Addison-Wesley Publishing Company, Menlo Park, Calif., 1987, pp. 7–13, 82–107.

Won Kim, "On Optimizing an SQL-like Nested Query," ACM Transactions on Database Systems, vol. 9, No. 3, Association for Computing Machinery, United States, Sep. 1982, pp. 443–469.

Barr & Feigenbau, Eds., *The Handbook of Artificial Intelligence,* vol. II, William Kaufmann, Inc., Los Altos, Calif., 1982, pp. 163–173.

Astrahan & Chamberlin, "Implementation of a Structured English Query Language," Communications of the ACM, United States, vol. 18, No. 10, Oct. 1975, pp. 580–588.

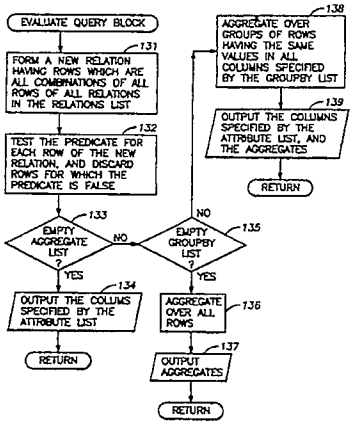
Primary Examiner—Paul V. Kulik

Attorney, Agent, or Firm—Blakely, Sokoloff, Taylor & Zafman

[57] **ABSTRACT**

The semantics of the outer join operator are extended to permit the application of different predicates to the join tuples and the anti-join tuples. For un-nesting of nested query blocks, the anti-join tuples, for example, are associated with a count value of zero instead of a count value of null. An inner query block is un-nested from an outer query block by converting the inner query to a first un-nested query generating a temporary relation and converting the outer query block to a second un-nested query receiving the precomputed temporary relation. When the nested inner query has an equi-join predicate joining a relation of the inner query to an outer query and a count aggregate, the query blocks are un-nested by removing the equi-join predicate from the inner query and placing a corresponding conjunctive (left) outerjoin predicate term in the predicate of the outer query, performing the count aggregate for each distinct value of the joining attribute of the relation of the inner query, and in the outer query applying different predicates to the joining and anti-joining tuples such that the predicate of the anti-joining tuples is evaluated assuming a count value of zero.

20 Claims, 18 Drawing Sheets



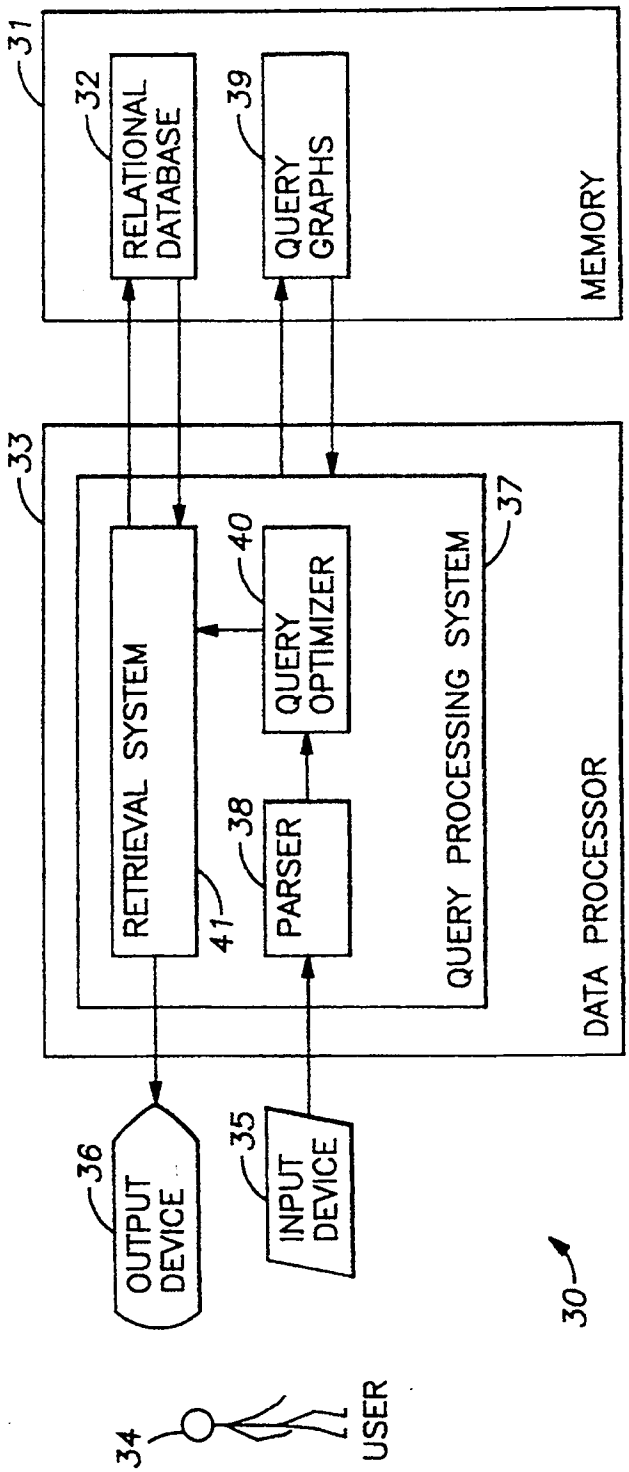

31
32
RELATIONAL DATABASE
39
QUERY GRAPHS
MEMORY
33
RETRIEVAL SYSTEM
40
QUERY OPTIMIZER
37
QUERY PROCESSING SYSTEM
38
PARSER
41
DATA PROCESSOR
36
OUTPUT DEVICE
35
INPUT DEVICE
30
34
USER

FIG. 1

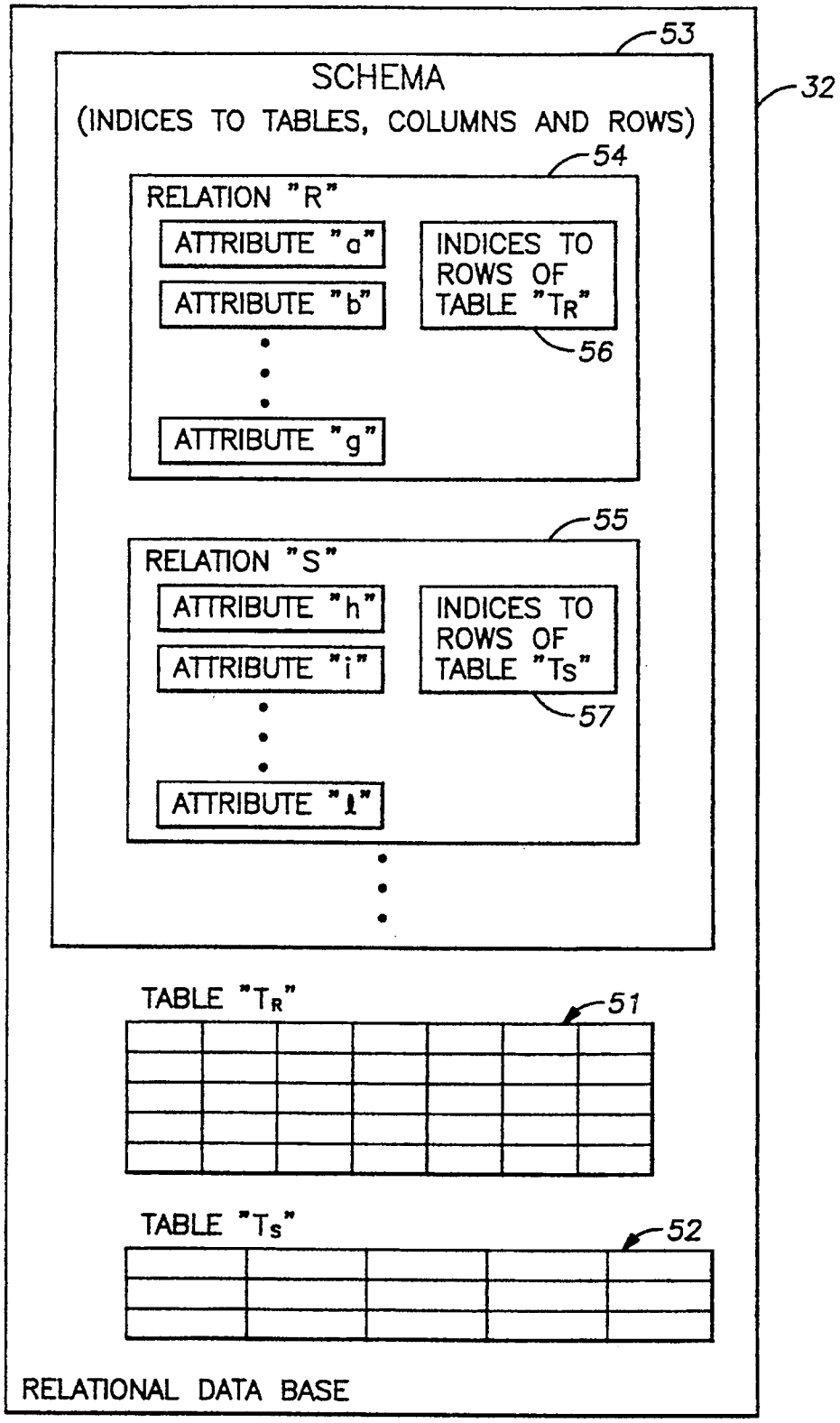

53
SCHEMA
(INDICES TO TABLES, COLUMNS AND ROWS)
32
54
RELATION "R"
ATTRIBUTE "a"
ATTRIBUTE "b"
ATTRIBUTE "g"
INDICES TO ROWS OF TABLE "TR"
56
55
RELATION "S"
ATTRIBUTE "h"
ATTRIBUTE "i"
ATTRIBUTE "l"
INDICES TO ROWS OF TABLE "Ts"
57
TABLE "TR"
51
TABLE "Ts"
52
RELATIONAL DATA BASE

FIG. 2

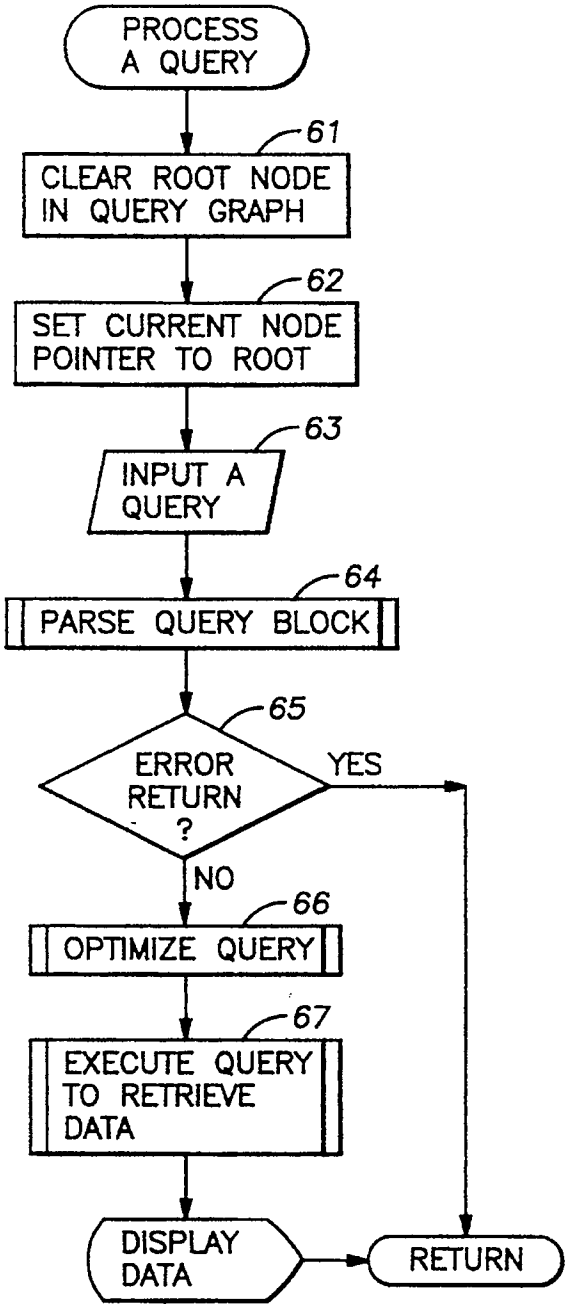

PROCESS A QUERY
61
CLEAR ROOT NODE IN QUERY GRAPH
62
SET CURRENT NODE POINTER TO ROOT
63
INPUT A QUERY
64
PARSE QUERY BLOCK
65
ERROR RETURN ?
YES
NO
66
OPTIMIZE QUERY
67
EXECUTE QUERY TO RETRIEVE DATA
DISPLAY DATA
RETURN

FIG. 3

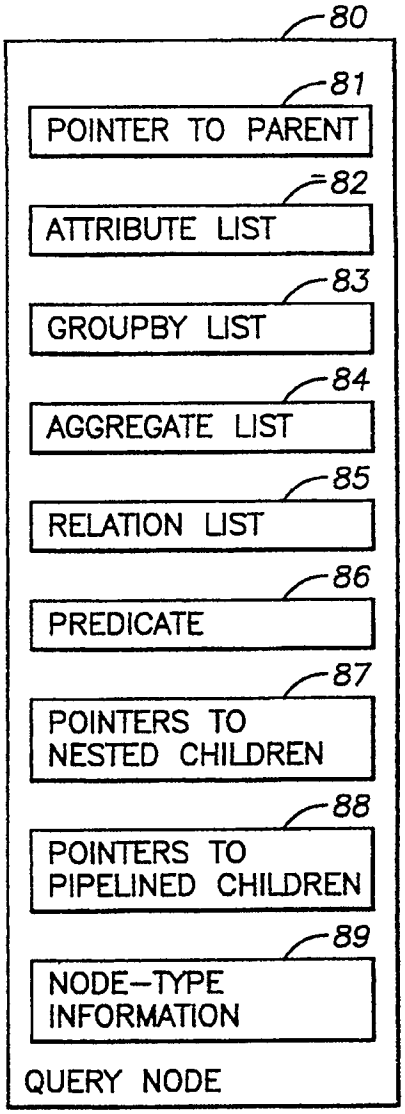

80
81
POINTER TO PARENT
82
ATTRIBUTE LIST
83
GROUPBY LIST
84
AGGREGATE LIST
85
RELATION LIST
86
PREDICATE
87
POINTERS TO NESTED CHILDREN
88
POINTERS TO PIPELINED CHILDREN
89
NODE-TYPE INFORMATION
QUERY NODE

FIG. 4

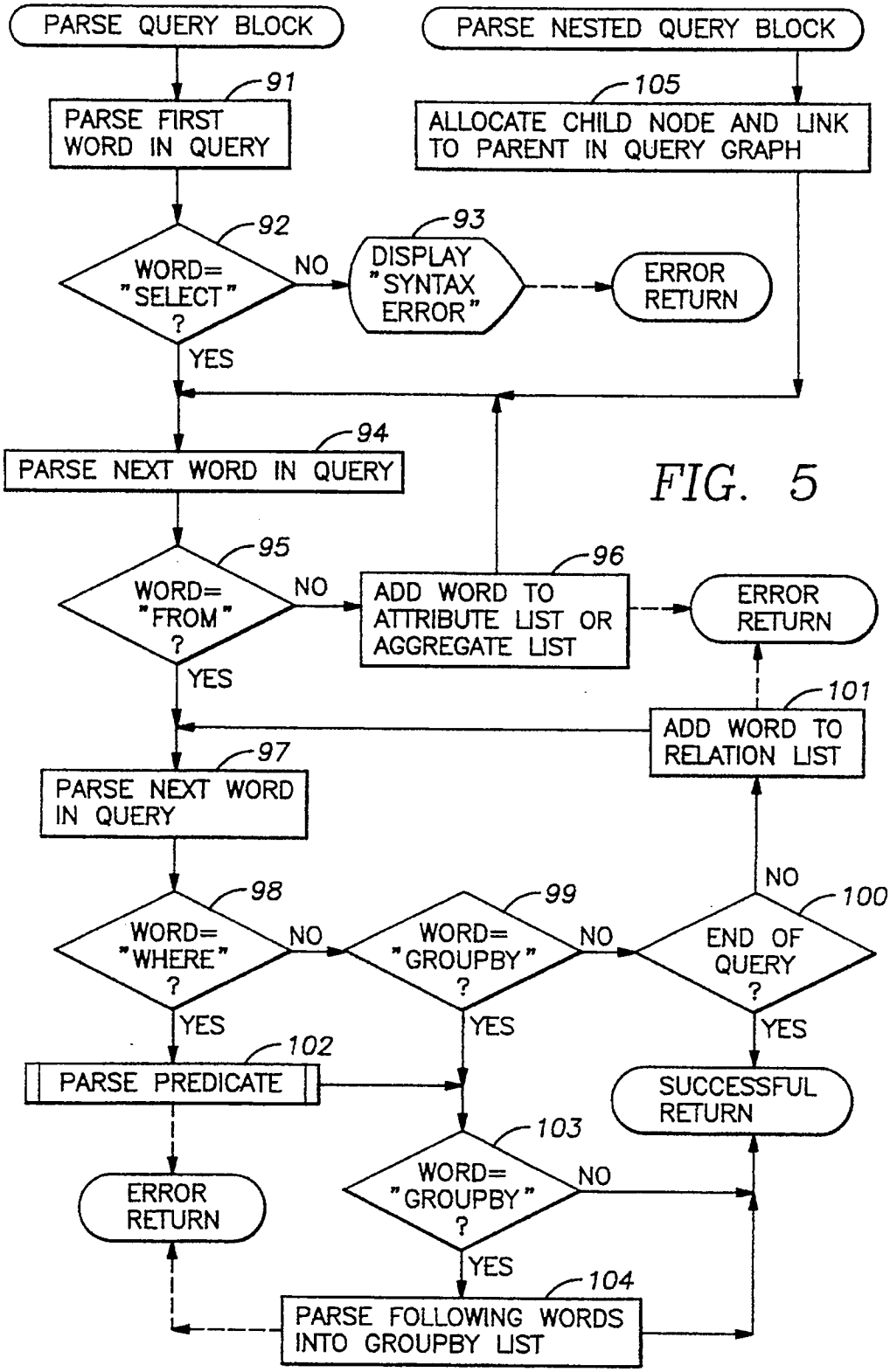

PARSE QUERY BLOCK
PARSE NESTED QUERY BLOCK
91
PARSE FIRST WORD IN QUERY
105
ALLOCATE CHILD NODE AND LINK TO PARENT IN QUERY GRAPH
92
WORD= "SELECT" ?
NO
93
DISPLAY "SYNTAX ERROR"
ERROR RETURN
YES
94
PARSE NEXT WORD IN QUERY
95
WORD= "FROM" ?
NO
96
ADD WORD TO ATTRIBUTE LIST OR AGGREGATE LIST
ERROR RETURN
YES
101
ADD WORD TO RELATION LIST
97
PARSE NEXT WORD IN QUERY
98
WORD= "WHERE" ?
NO
99
WORD= "GROUPBY" ?
NO
100
END OF QUERY ?
NO
YES
YES
YES
102
PARSE PREDICATE
SUCCESSFUL RETURN
103
WORD= "GROUPBY" ?
NO
ERROR RETURN
YES
104
PARSE FOLLOWING WORDS INTO GROUPBY LIST

FIG. 5

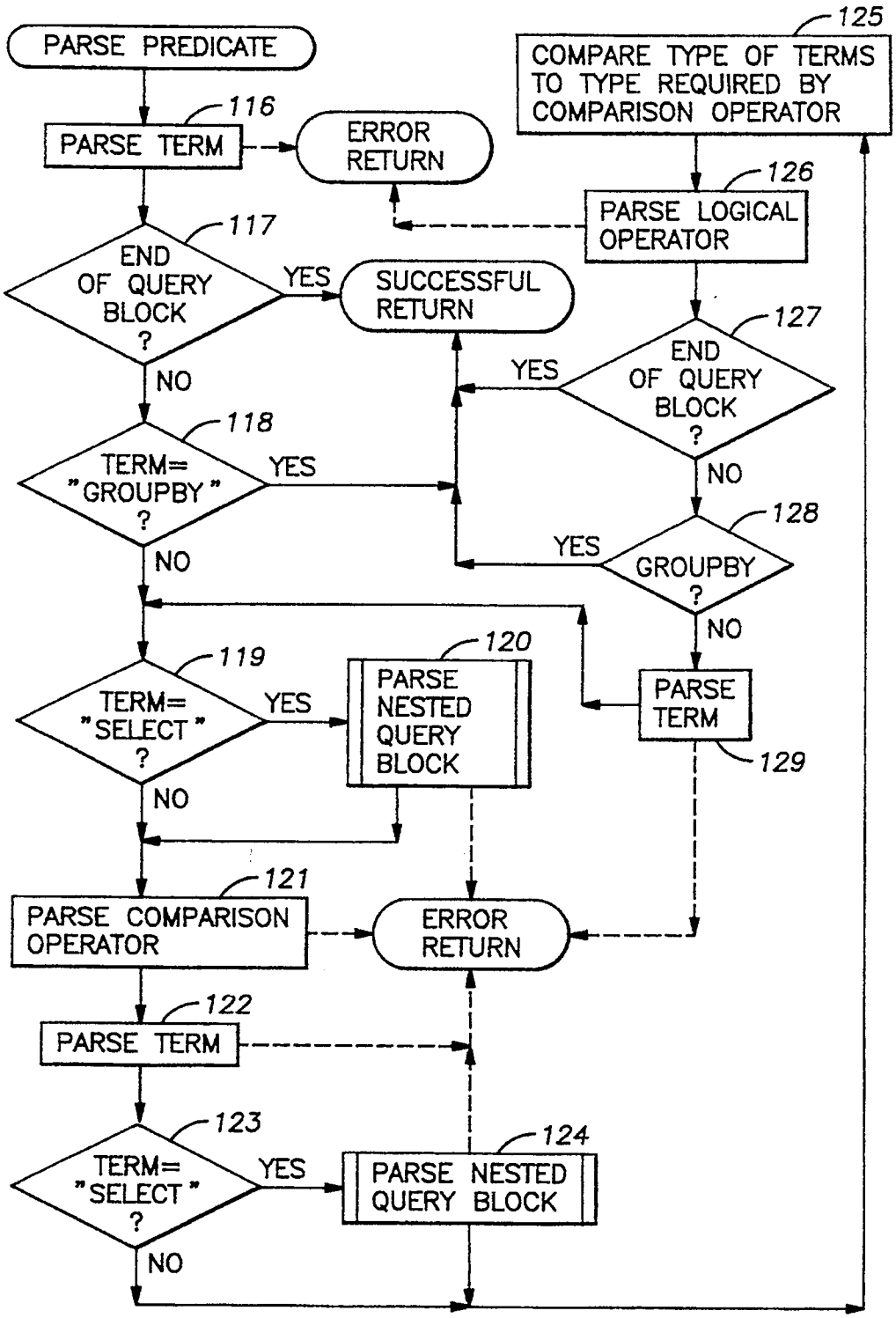

PARSE PREDICATE
116
PARSE TERM
ERROR RETURN
117
END OF QUERY BLOCK ?
YES
SUCCESSFUL RETURN
NO
118
TERM= "GROUPBY" ?
YES
NO
119
TERM= "SELECT" ?
YES
120
PARSE NESTED QUERY BLOCK
NO
121
PARSE COMPARISON OPERATOR
ERROR RETURN
122
PARSE TERM
123
TERM= "SELECT" ?
YES
124
PARSE NESTED QUERY BLOCK
NO
125
COMPARE TYPE OF TERMS TO TYPE REQUIRED BY COMPARISON OPERATOR
126
PARSE LOGICAL OPERATOR
127
END OF QUERY BLOCK ?
YES
NO
128
GROUPBY ?
YES
NO
PARSE TERM
129

FIG. 6

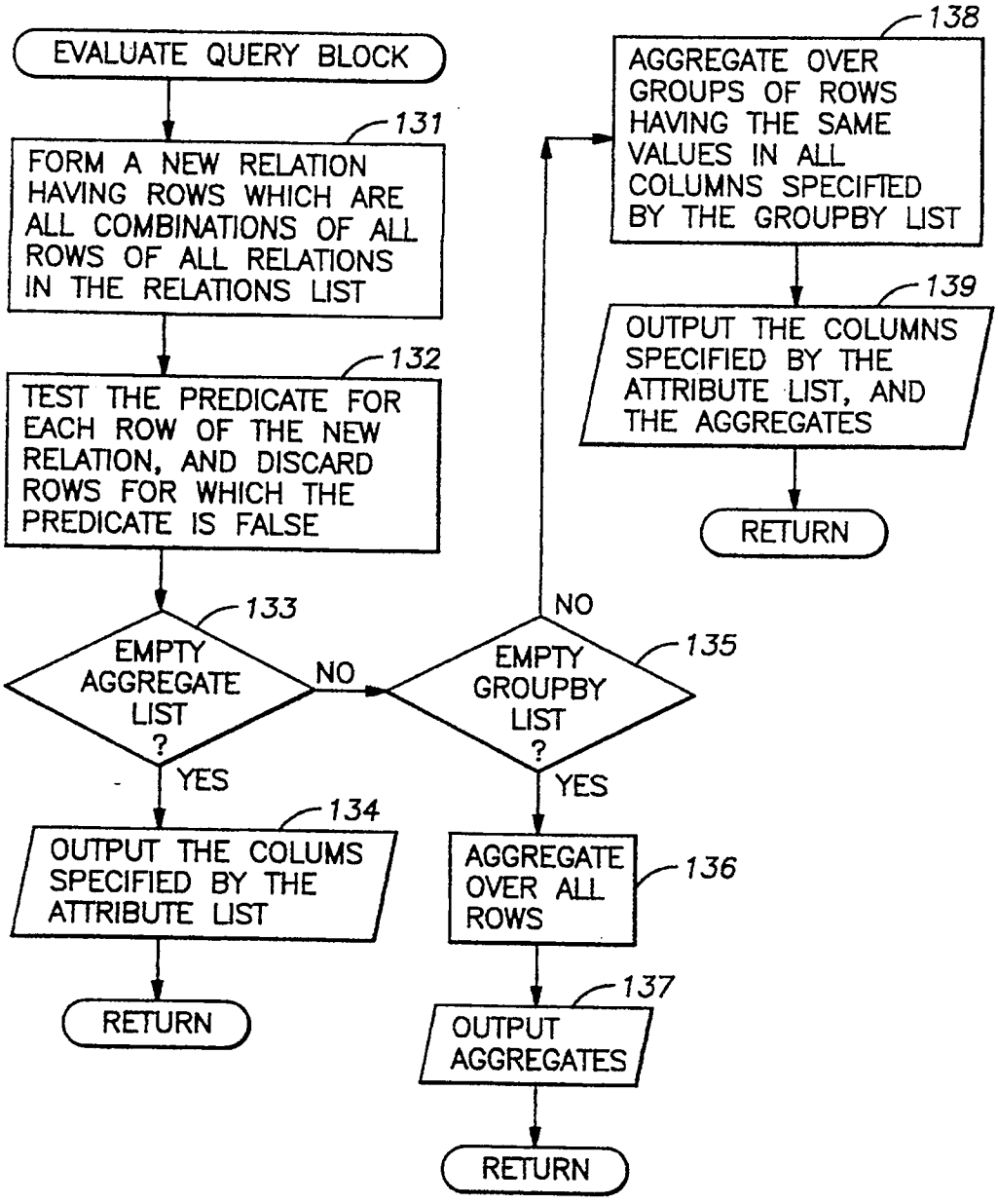

EVALUATE QUERY BLOCK
131
FORM A NEW RELATION HAVING ROWS WHICH ARE ALL COMBINATIONS OF ALL ROWS OF ALL RELATIONS IN THE RELATIONS LIST
132
TEST THE PREDICATE FOR EACH ROW OF THE NEW RELATION, AND DISCARD ROWS FOR WHICH THE PREDICATE IS FALSE
133
EMPTY AGGREGATE LIST ?
NO
YES
134
OUTPUT THE COLUMS SPECIFIED BY THE ATTRIBUTE LIST
RETURN
135
EMPTY GROUPBY LIST ?
NO
YES
136
AGGREGATE OVER ALL ROWS
137
OUTPUT AGGREGATES
RETURN
138
AGGREGATE OVER GROUPS OF ROWS HAVING THE SAME VALUES IN ALL COLUMNS SPECIFIED BY THE GROUPBY LIST
139
OUTPUT THE COLUMNS SPECIFIED BY THE ATTRIBUTE LIST, AND THE AGGREGATES
RETURN

FIG. 7

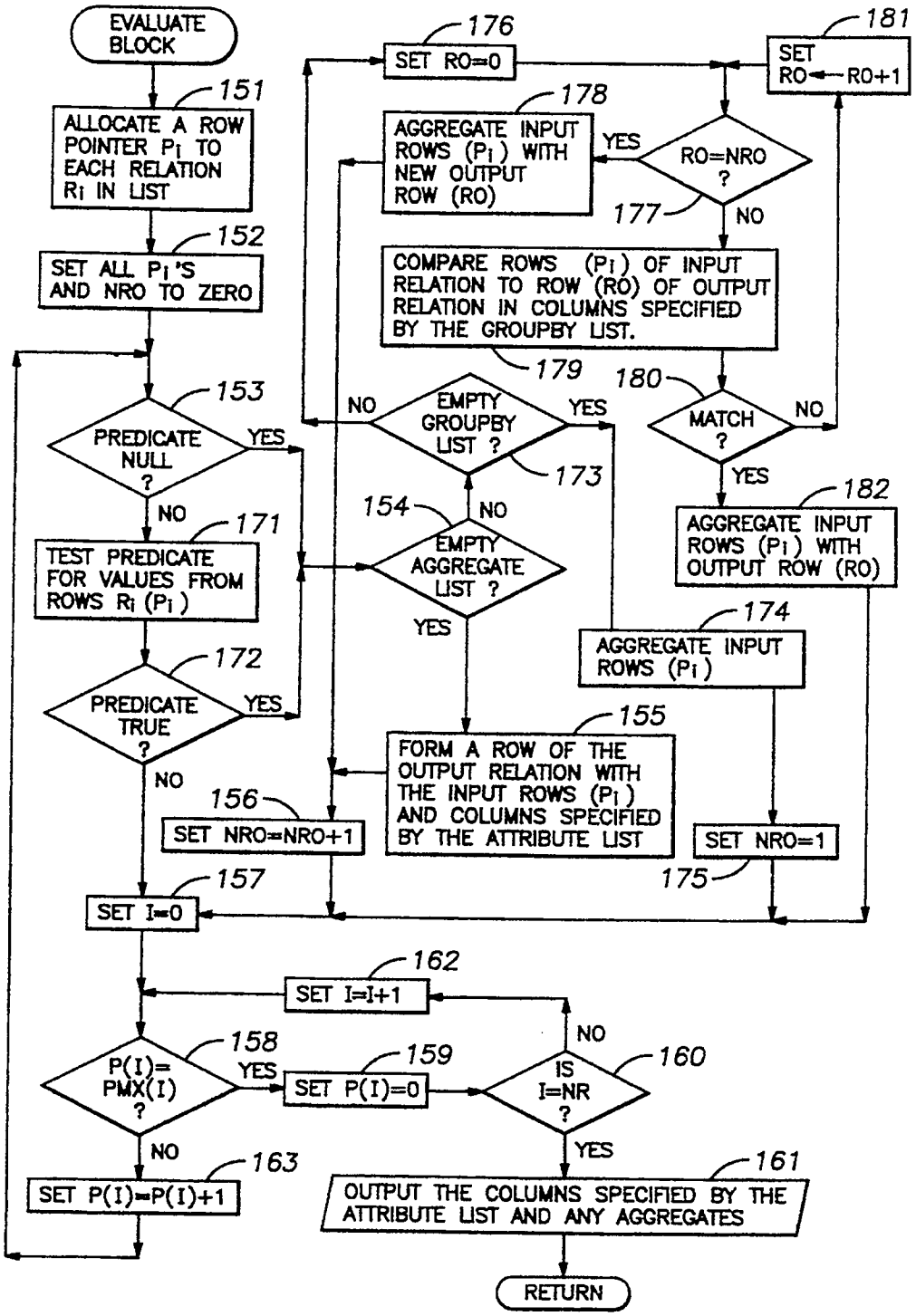

EVALUATE BLOCK
151
ALLOCATE A ROW POINTER P_I TO EACH RELATION R_I IN LIST
152
SET ALL P_I'S AND NRO TO ZERO
153
PREDICATE NULL ?
YES
NO
171
TEST PREDICATE FOR VALUES FROM ROWS R_I (P_I)
172
PREDICATE TRUE ?
YES
NO
156
SET NRO=NRO+1
157
SET I=0
158
P(I)= PMX(I) ?
YES
NO
163
SET P(I)=P(I)+1
162
SET I=I+1
159
SET P(I)=0
160
IS I=NR ?
NO
YES
161
OUTPUT THE COLUMNS SPECIFIED BY THE ATTRIBUTE LIST AND ANY AGGREGATES
RETURN
154
EMPTY AGGREGATE LIST ?
YES
NO
155
FORM A ROW OF THE OUTPUT RELATION WITH THE INPUT ROWS (P_I) AND COLUMNS SPECIFIED BY THE ATTRIBUTE LIST
173
EMPTY GROUPBY LIST ?
NO
YES
174
AGGREGATE INPUT ROWS (P_I)
175
SET NRO=1
176
SET RO=0
178
AGGREGATE INPUT ROWS (P_I) WITH NEW OUTPUT ROW (RO)
177
RO=NRO ?
YES
NO
179
COMPARE ROWS (P_I) OF INPUT RELATION TO ROW (RO) OF OUTPUT RELATION IN COLUMNS SPECIFIED BY THE GROUPBY LIST.
180
MATCH ?
NO
YES
181
SET RO←RO+1
182
AGGREGATE INPUT ROWS (P_I) WITH OUTPUT ROW (RO)

FIG. 8

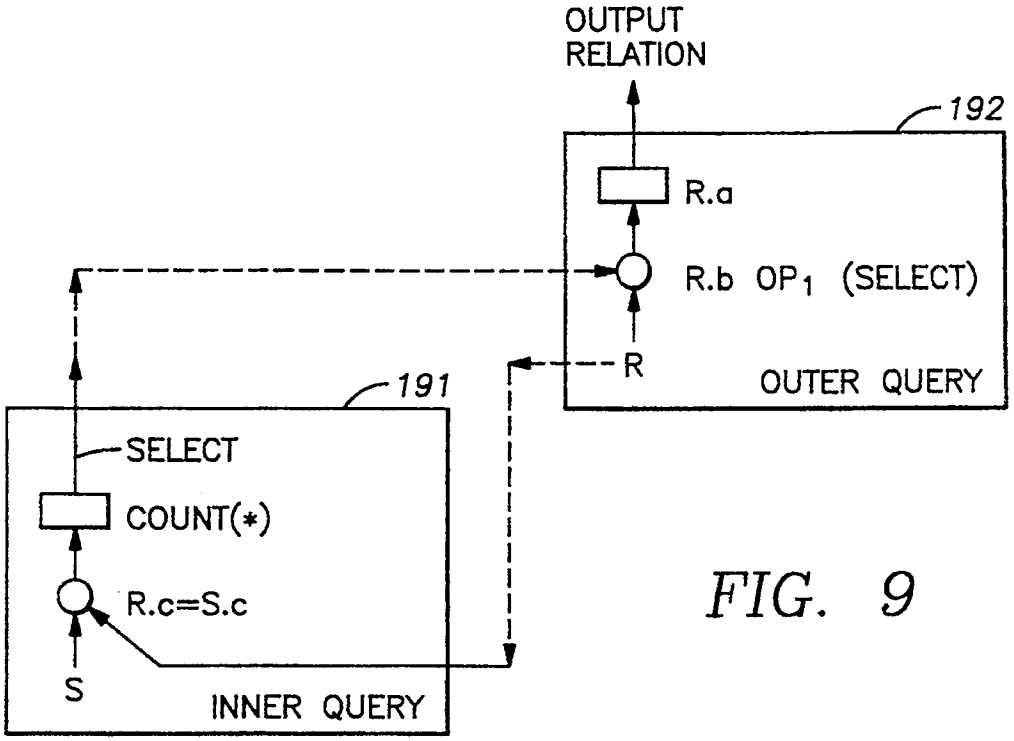

OUTPUT
RELATION
192
R.a
R.b OP1 (SELECT)
R
OUTER QUERY
191
SELECT
COUNT(*)
R.c=S.c
S
INNER QUERY

FIG. 9

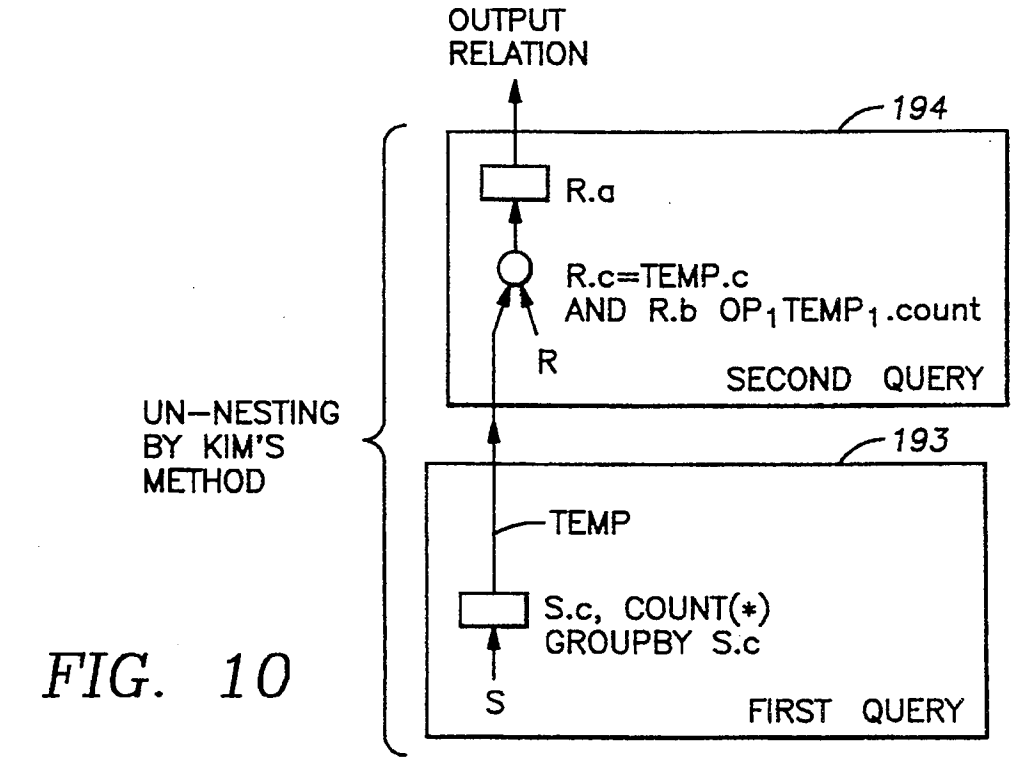

OUTPUT
RELATION
194
R.a
R.c=TEMP.c
AND R.b OP1 TEMP1.count
R
SECOND QUERY
UN-NESTING
BY KIM'S
METHOD
193
TEMP
S.c, COUNT(*)
GROUPBY S.c
S
FIRST QUERY

FIG. 10

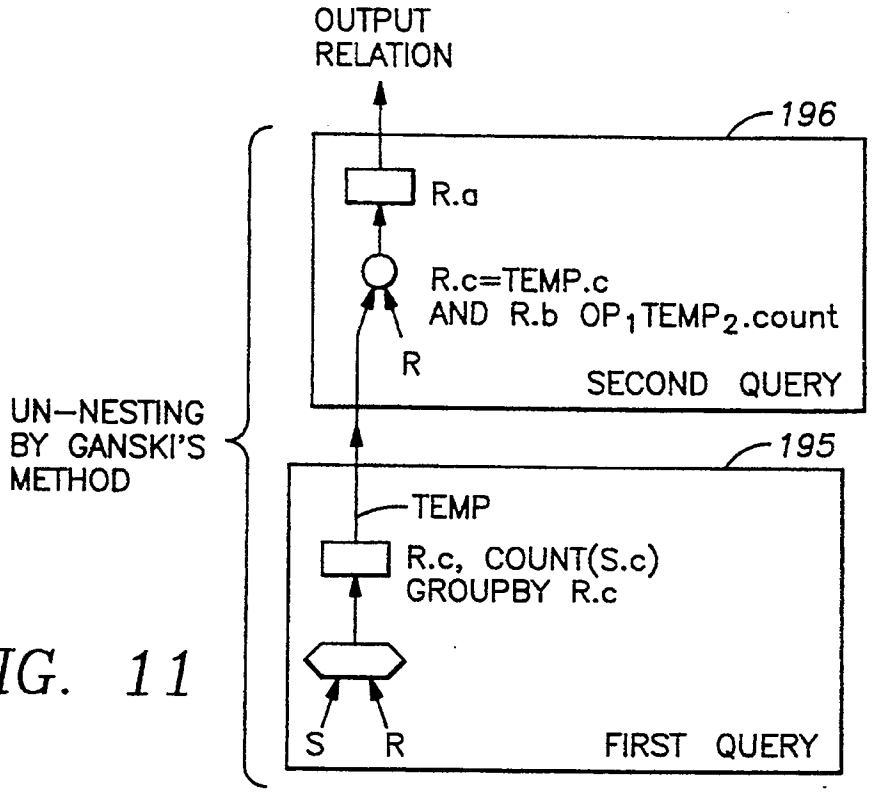

OUTPUT RELATION
196
R.a
R.c=TEMP.c AND R.b OP1TEMP2.count
R
SECOND QUERY
UN–NESTING BY GANSKI'S METHOD
195
TEMP
R.c, COUNT(S.c) GROUPBY R.c
S
R
FIRST QUERY

FIG. 11

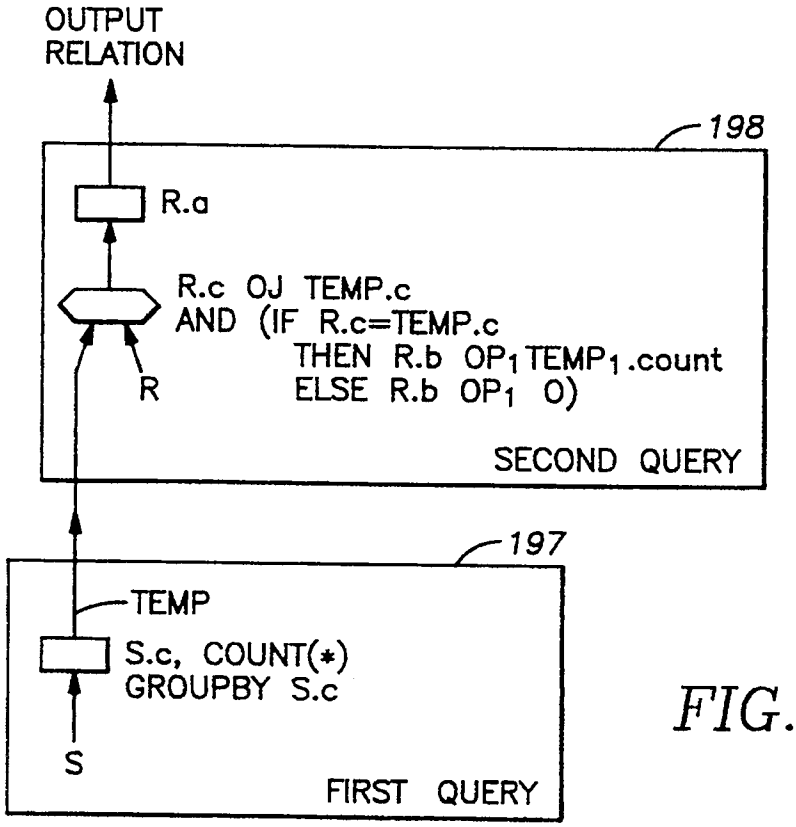

OUTPUT RELATION
198
R.a
R.c OJ TEMP.c AND (IF R.c=TEMP.c THEN R.b OP1 TEMP1.count ELSE R.b OP1 0)
R
SECOND QUERY
197
TEMP
S.c, COUNT(*) GROUPBY S.c
S
FIRST QUERY

FIG. 12

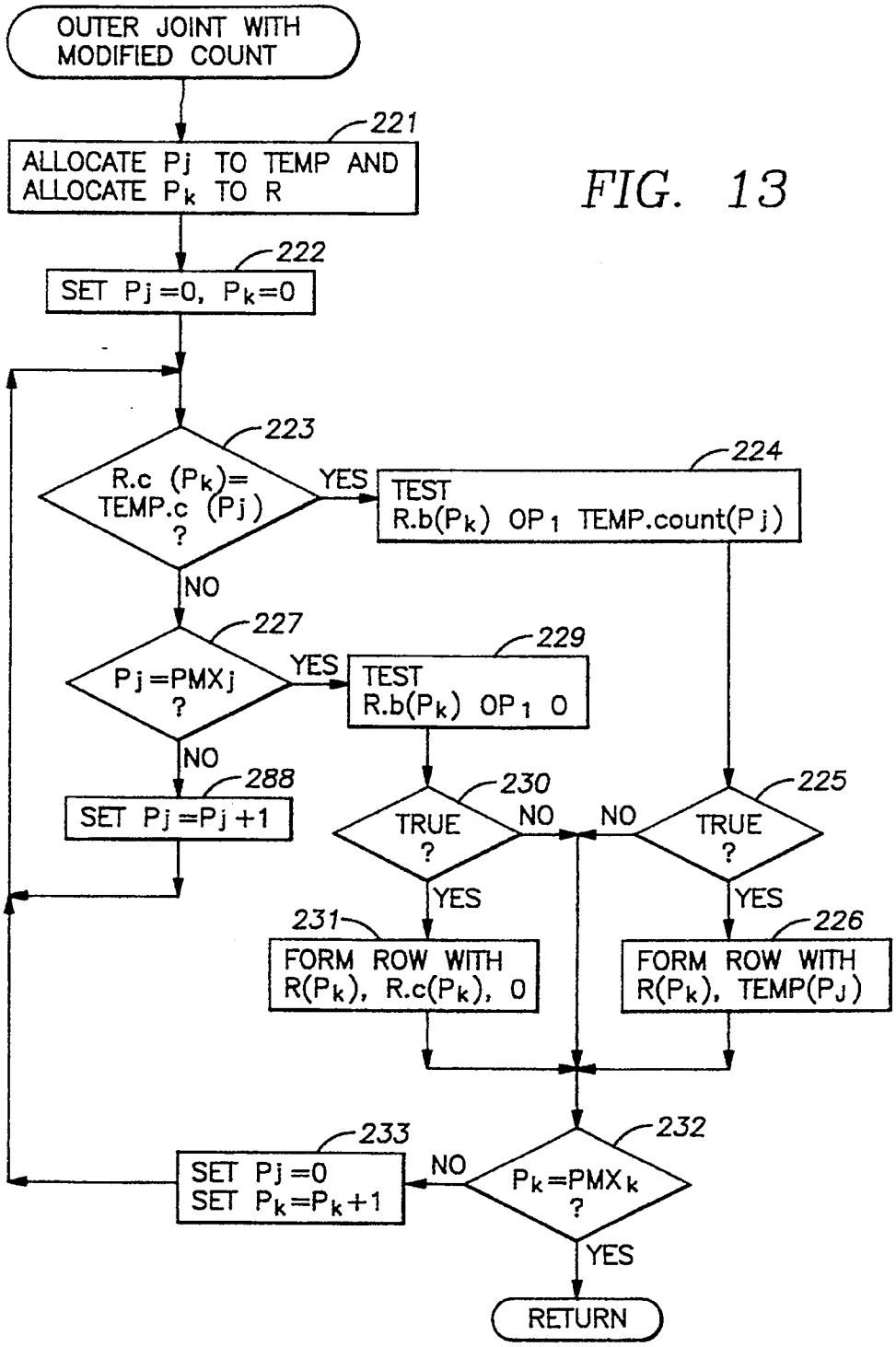
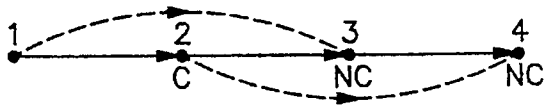

OUTER JOINT WITH MODIFIED COUNT
221
ALLOCATE Pj TO TEMP AND ALLOCATE Pk TO R
222
SET Pj=0, Pk=0
223
R.c (Pk)= TEMP.c (Pj) ?
YES
224
TEST R.b(Pk) OP1 TEMP.count(Pj)
NO
227
Pj=PMXj ?
YES
229
TEST R.b(Pk) OP1 0
NO
288
SET Pj=Pj+1
230
TRUE ?
NO
NO
225
TRUE ?
YES
YES
231
FORM ROW WITH R(Pk), R.c(Pk), 0
226
FORM ROW WITH R(Pk), TEMP(PJ)
233
SET Pj=0 SET Pk=Pk+1
NO
232
Pk=PMXk ?
YES
RETURN

FIG. 13

FIG. 14


1
2
3
4
C
NC
NC

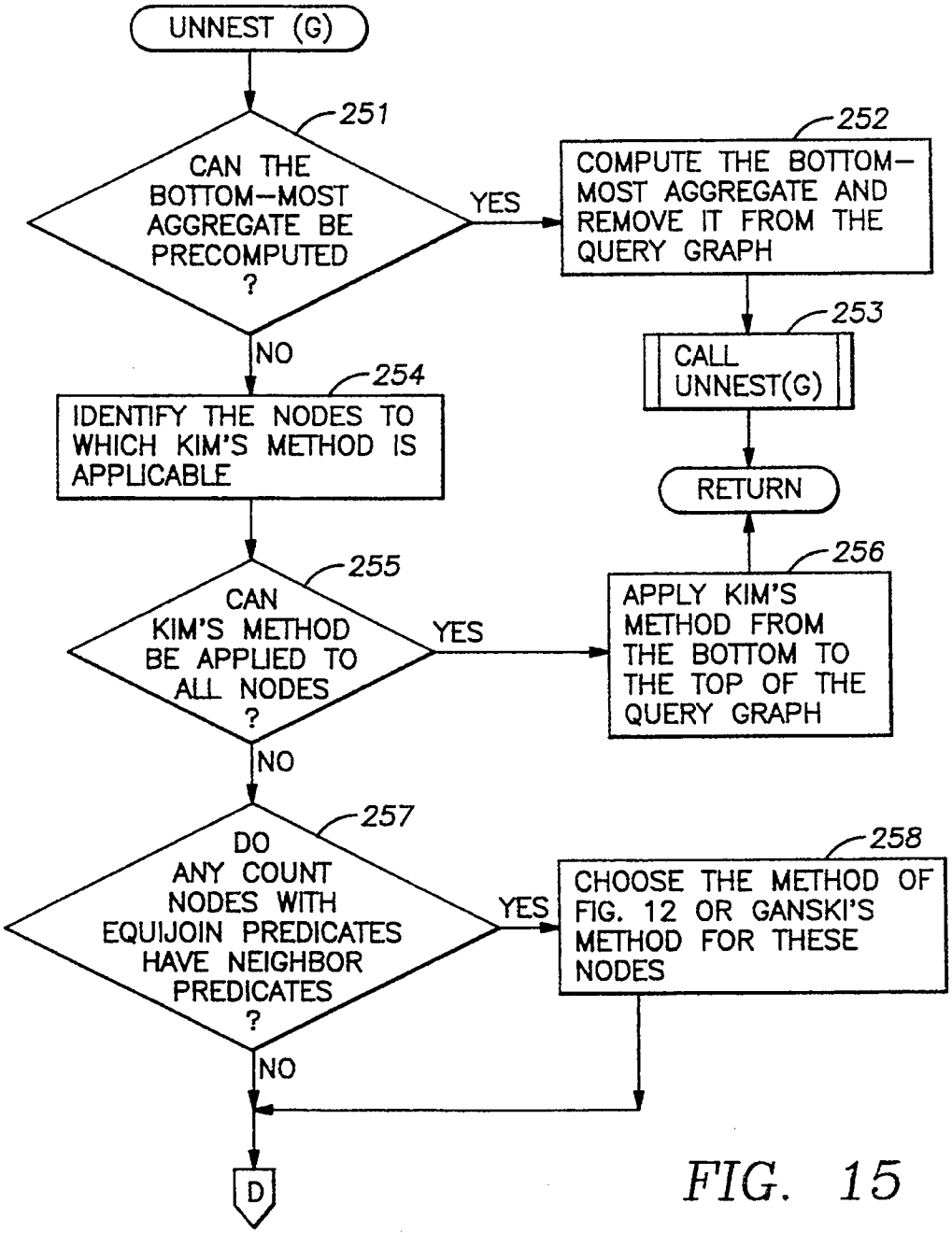

UNNEST (G)
251
CAN THE BOTTOM–MOST AGGREGATE BE PRECOMPUTED ?
YES
NO
252
COMPUTE THE BOTTOM–MOST AGGREGATE AND REMOVE IT FROM THE QUERY GRAPH
253
CALL UNNEST(G)
RETURN
254
IDENTIFY THE NODES TO WHICH KIM'S METHOD IS APPLICABLE
255
CAN KIM'S METHOD BE APPLIED TO ALL NODES ?
YES
256
APPLY KIM'S METHOD FROM THE BOTTOM TO THE TOP OF THE QUERY GRAPH
NO
257
DO ANY COUNT NODES WITH EQUIJOIN PREDICATES HAVE NEIGHBOR PREDICATES ?
YES
258
CHOOSE THE METHOD OF FIG. 12 OR GANSKI'S METHOD FOR THESE NODES
NO
D

FIG. 15

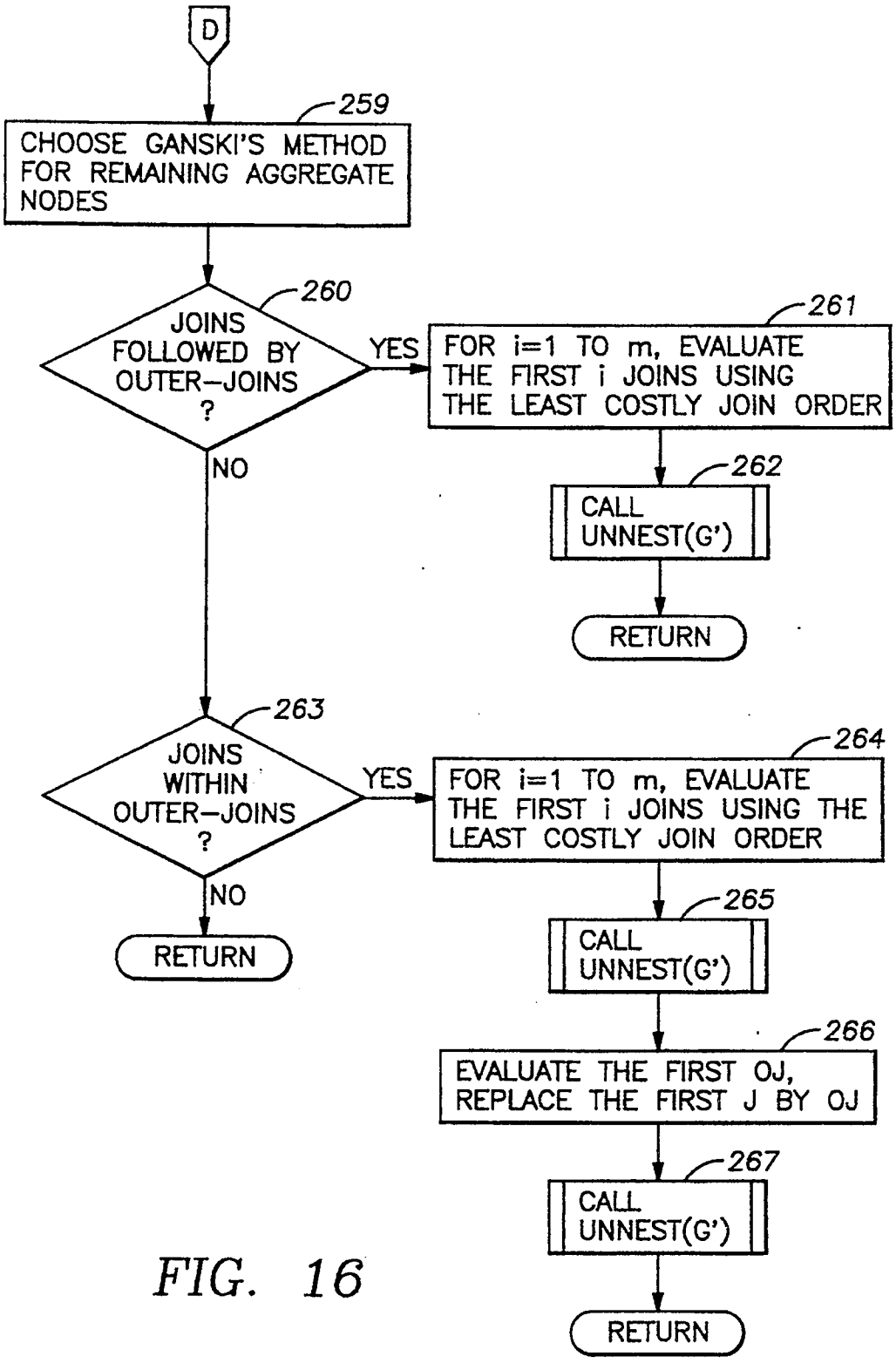

D
259
CHOOSE GANSKI'S METHOD FOR REMAINING AGGREGATE NODES
260
JOINS FOLLOWED BY OUTER–JOINS ?
YES
261
FOR i=1 TO m, EVALUATE THE FIRST i JOINS USING THE LEAST COSTLY JOIN ORDER
262
CALL UNNEST(G')
RETURN
NO
263
JOINS WITHIN OUTER–JOINS ?
YES
264
FOR i=1 TO m, EVALUATE THE FIRST i JOINS USING THE LEAST COSTLY JOIN ORDER
265
CALL UNNEST(G')
266
EVALUATE THE FIRST OJ, REPLACE THE FIRST J BY OJ
267
CALL UNNEST(G')
RETURN
NO
RETURN

FIG. 16

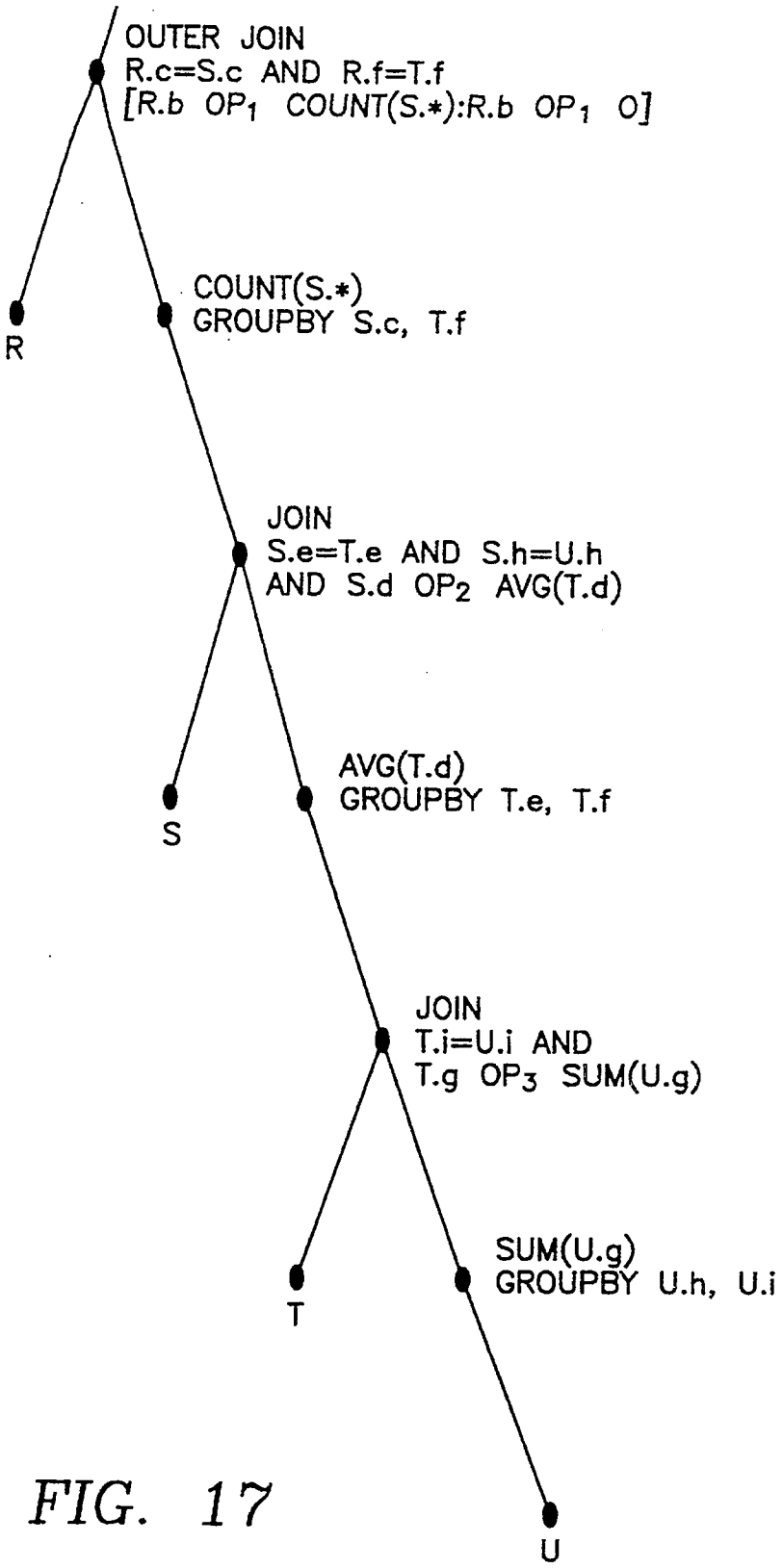

OUTER JOIN
R.c=S.c AND R.f=T.f
[R.b OP1 COUNT(S.*):R.b OP1 0]
COUNT(S.*)
GROUPBY S.c, T.f
R
JOIN
S.e=T.e AND S.h=U.h
AND S.d OP2 AVG(T.d)
AVG(T.d)
GROUPBY T.e, T.f
S
JOIN
T.i=U.i AND
T.g OP3 SUM(U.g)
SUM(U.g)
GROUPBY U.h, U.i
T
U

FIG. 17

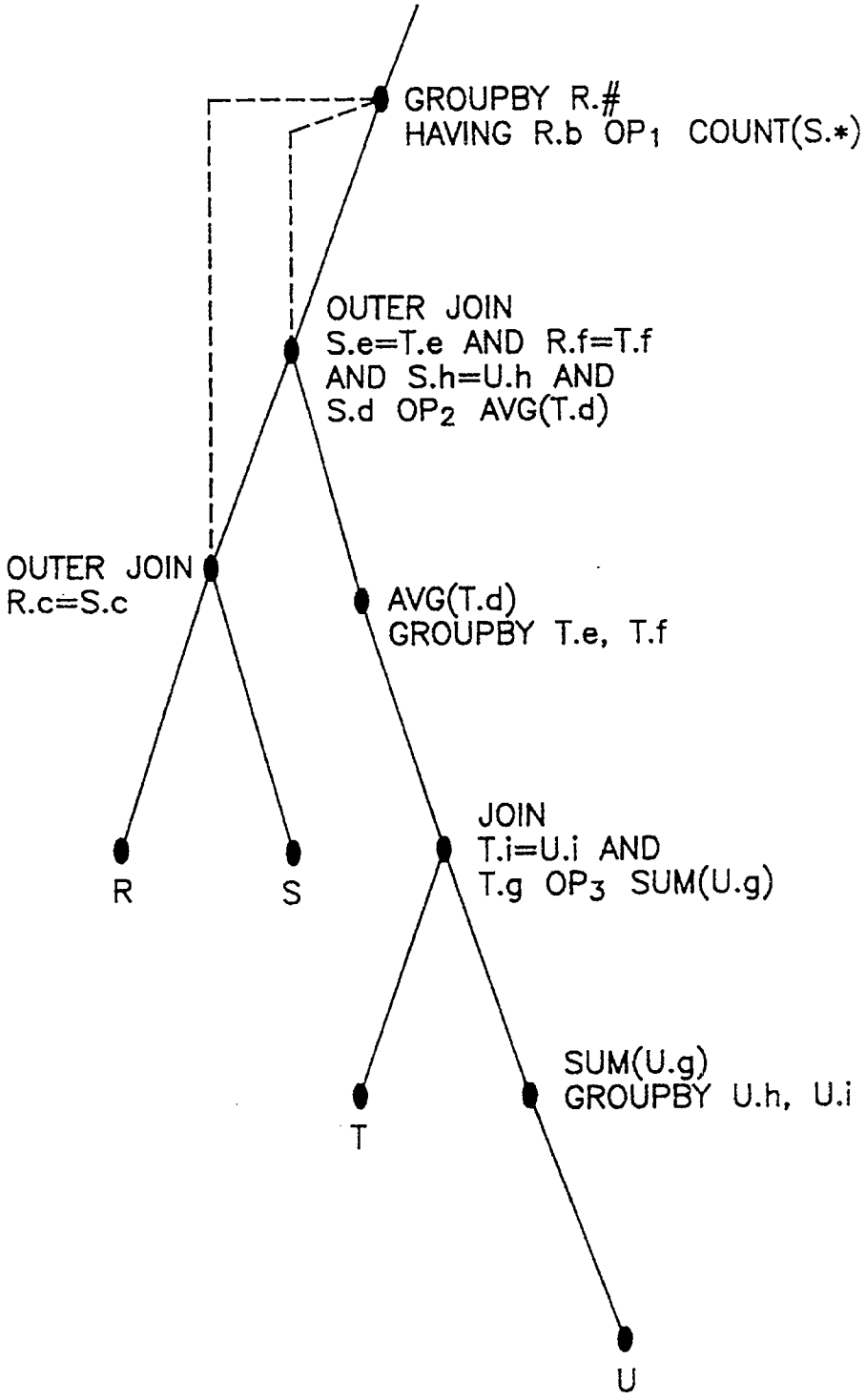

GROUPBY R.#
HAVING R.b OP1 COUNT(S.*)
OUTER JOIN
S.e=T.e AND R.f=T.f
AND S.h=U.h AND
S.d OP2 AVG(T.d)
OUTER JOIN
R.c=S.c
AVG(T.d)
GROUPBY T.e, T.f
JOIN
T.i=U.i AND
T.g OP3 SUM(U.g)
R
S
T
SUM(U.g)
GROUPBY U.h, U.i
U

FIG. 18

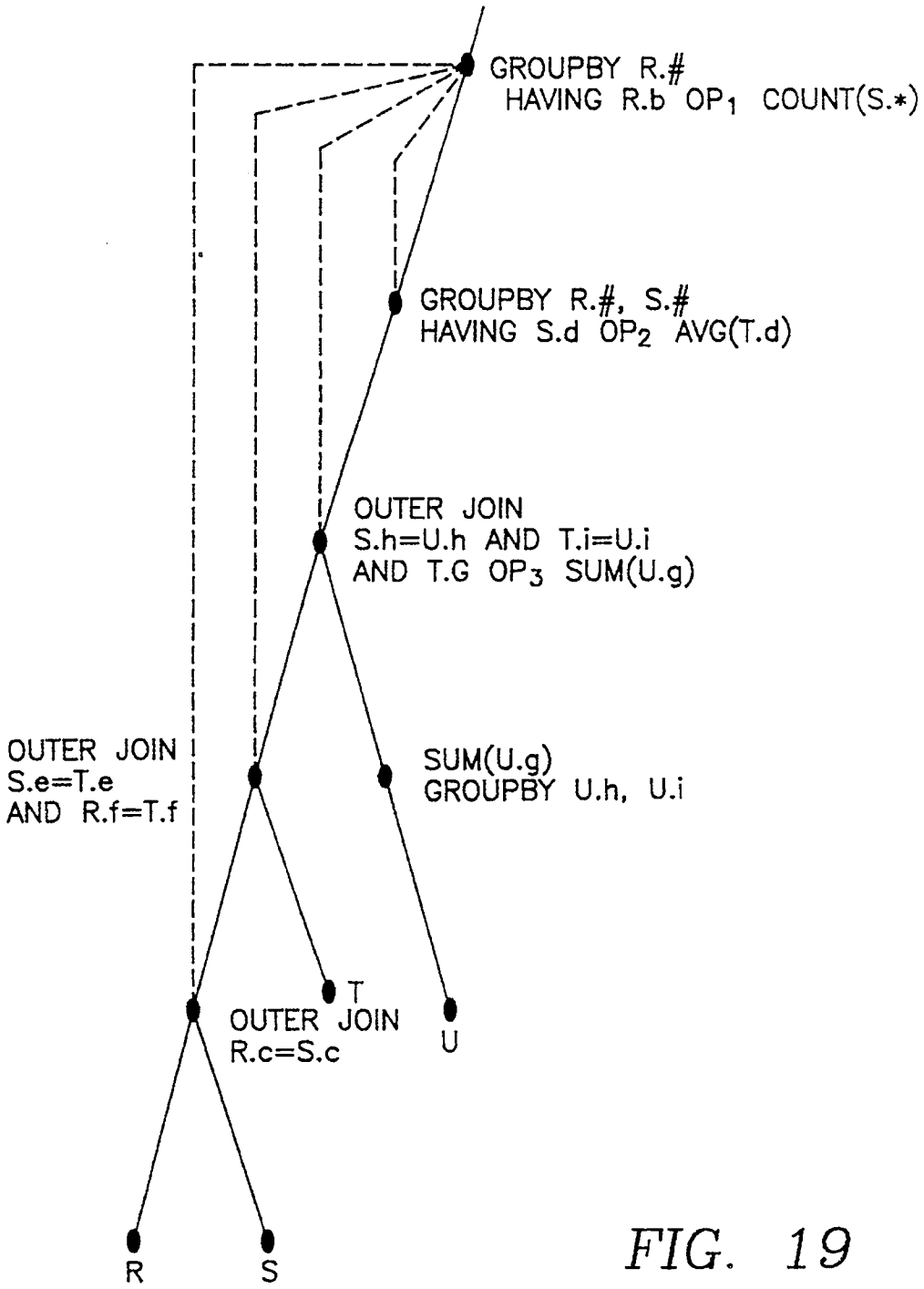

GROUPBY R.#
HAVING R.b OP1 COUNT(S.*)
GROUPBY R.#, S.#
HAVING S.d OP2 AVG(T.d)
OUTER JOIN
S.h=U.h AND T.i=U.i
AND T.G OP3 SUM(U.g)
OUTER JOIN
S.e=T.e
AND R.f=T.f
SUM(U.g)
GROUPBY U.h, U.i
T
OUTER JOIN
R.c=S.c
U
R
S

FIG. 19

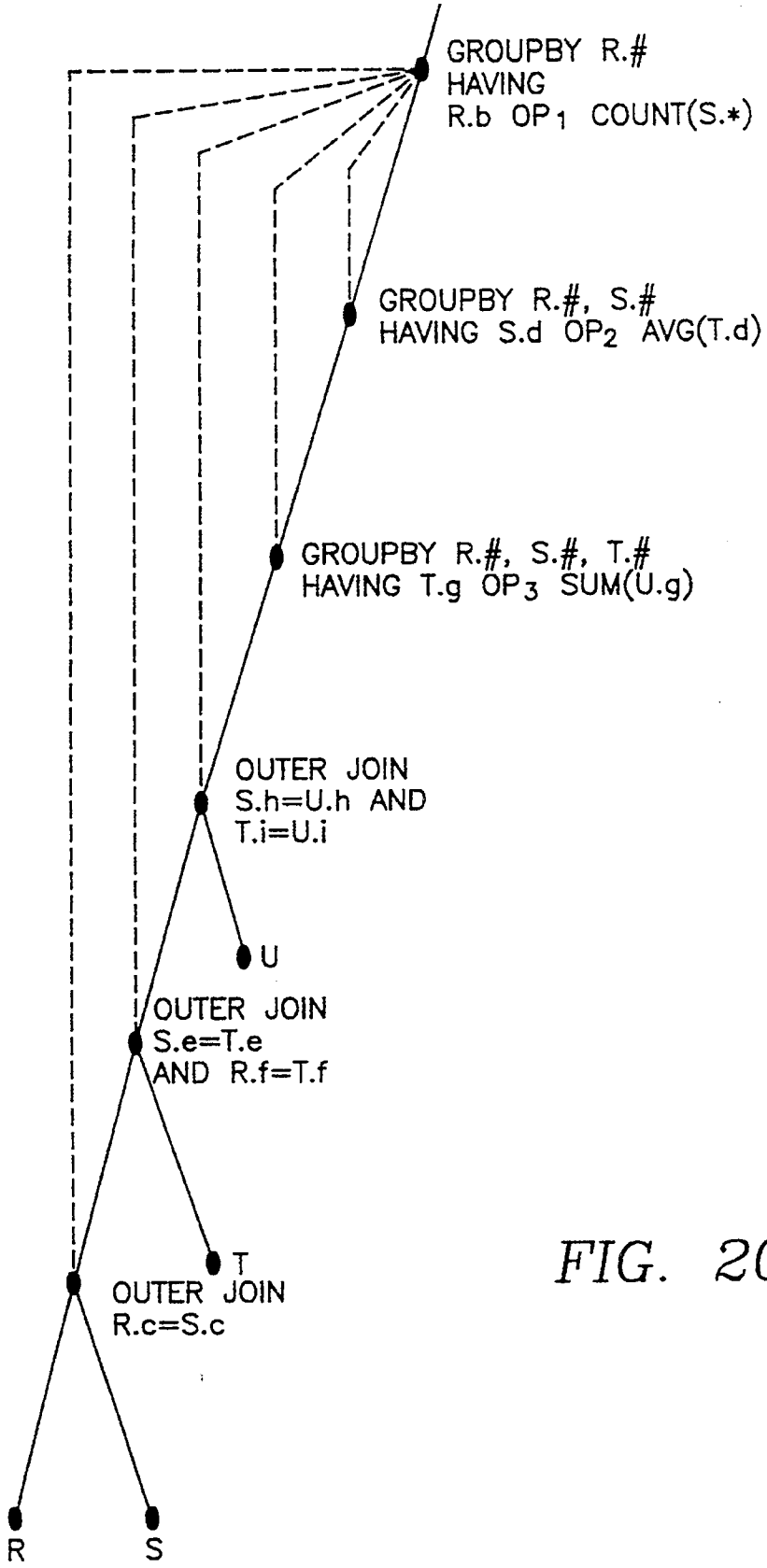

GROUPBY R.#
HAVING
R.b OP1 COUNT(S.*)
GROUPBY R.#, S.#
HAVING S.d OP2 AVG(T.d)
GROUPBY R.#, S.#, T.#
HAVING T.g OP3 SUM(U.g)
OUTER JOIN
S.h=U.h AND
T.i=U.i
U
OUTER JOIN
S.e=T.e
AND R.f=T.f
T
OUTER JOIN
R.c=S.c
R
S

FIG. 20

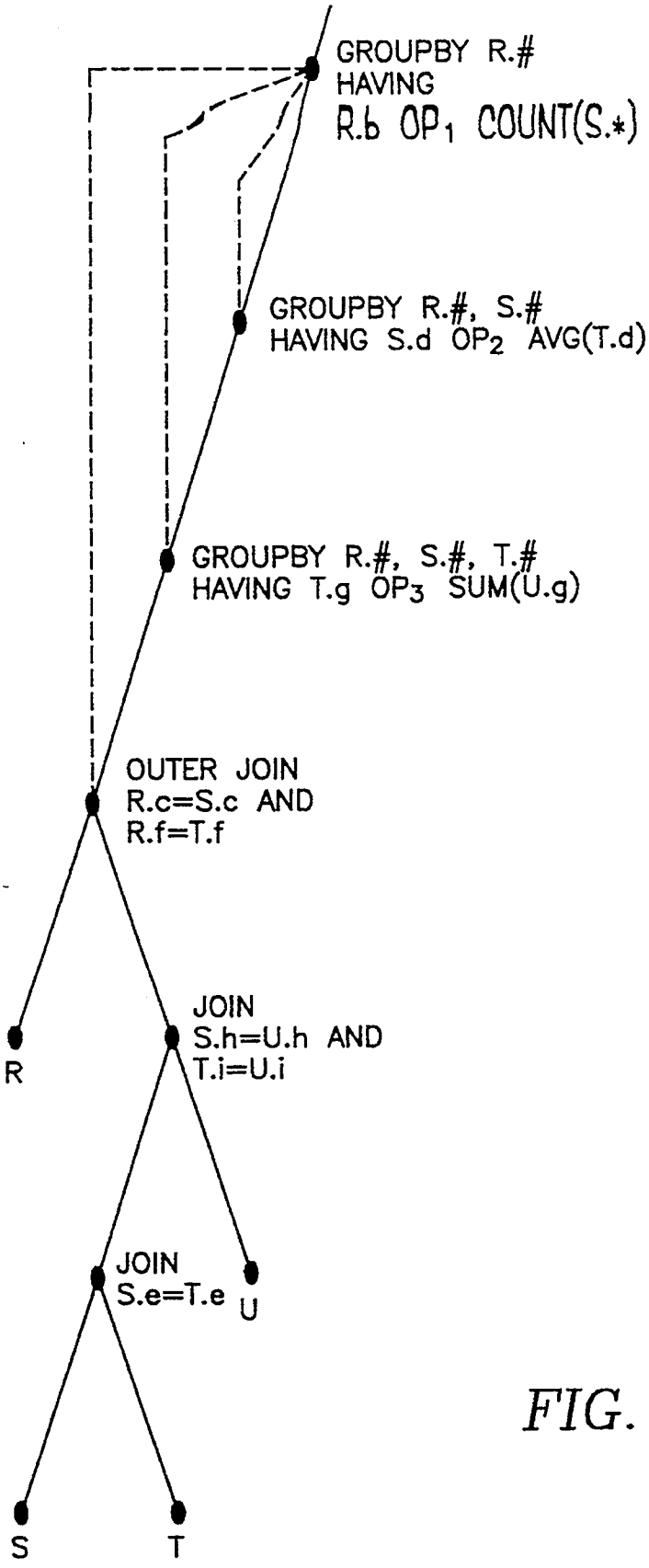

GROUPBY R.#
HAVING
R.b OP1 COUNT(S.*)
GROUPBY R.#, S.#
HAVING S.d OP2 AVG(T.d)
GROUPBY R.#, S.#, T.#
HAVING T.g OP3 SUM(U.g)
OUTER JOIN
R.c=S.c AND
R.f=T.f
JOIN
S.h=U.h AND
T.i=U.i
R
JOIN
S.e=T.e
U
S
T

FIG. 21

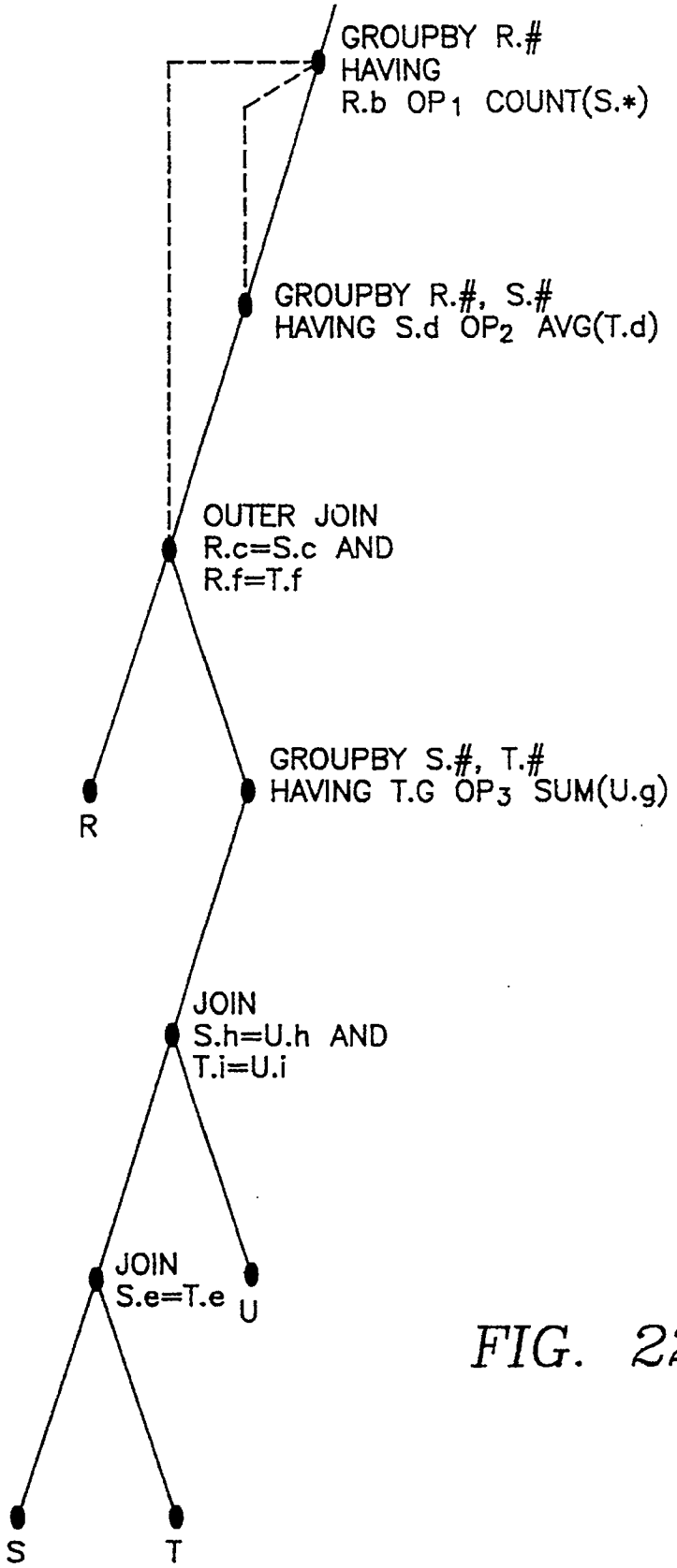

GROUPBY R.#
HAVING
R.b OP1 COUNT(S.*)
GROUPBY R.#, S.#
HAVING S.d OP2 AVG(T.d)
OUTER JOIN
R.c=S.c AND
R.f=T.f
R
GROUPBY S.#, T.#
HAVING T.G OP3 SUM(U.g)
JOIN
S.h=U.h AND
T.i=U.i
JOIN
S.e=T.e
U
S
T

FIG. 22

EXTENDING THE SEMANTICS OF THE OUTER JOIN OPERATOR FOR UN-NESTING QUERIES TO A DATA BASE

BACKGROUND OF THE INVENTION

1. Technical Field

The present invention relates generally to database management systems, and more particularly to query optimization in a database system. The present invention specifically relates to un-nesting nested database queries by distinguishing between joining and anti-joining tuples during an outer join operation.

2. Background of the Invention

A database management system is a computer system that provides for the storage and retrieval of information about a subject domain. Typical examples are airline reservation systems, payroll systems, and inventory systems. The database management system includes an organized collection of data about the subject domain, and a data-manipulation language for querying and altering the data. Typically, the data are organized as "tuples" of respective values for predefined attributes.

In a so-called "relational" database management system, the tuples of data are stored in a plurality of tables, each of which corresponds to a set of tuples for common predefined attributes. Each table corresponds to a single "relation". Each attribute of the relation corresponds to a column of the table, and each tuple of values corresponds to a particular row of the table. Each value in the table corresponds to a particular row or tuple and a particular column or attribute. A collection of related tables or relations in the database is known as a schema.

The data manipulation language for a relational database management system typically specifies operations upon one or more relations to create a new relation. A "restriction" operation forms a subset of the rows in a table by applying a condition or "predicate" to the values in each row. A "projection" operation removes columns from the rows by forming a stream of rows with only specified columns. A "join" operation combines data from a first table with data from a second table, typically by applying a predicate comparing the values in a column of the first table to the values in a related column of the second table. Usually such a join predicate is an "equi-join" predicate in which a first function of the attribute corresponding to the column of the first table must equal a second function of the attribute corresponding to the column of the second table.

Yet another operation upon a relation is known as an aggregation, in which a new column of values is generated by combining or aggregating values from all of the rows, or specified groups of rows. Aggregate functions include, for example, a count of the rows in each group, or the sum of the values of a specified attribute for all rows in each group. Typically, the rows are grouped for aggregation such that the rows in each group have equal values for a specified column or attribute. Therefore, a new aggregate value is generated corresponding to each distinct value of the specified attribute.

The query language for a relational database system typically defines a syntax for specifying a "query block" including a list of relations to be accessed, a predicate to govern restriction or join operations, a list of attributes to specify a projection operation, and a list of aggregate functions. If at least one attribute is specified and at least one aggregate function is also specified, then a meaningful result usually would require the aggregate to be grouped for each distinct value of the specified attribute. Although such a grouping could be presumed, the query language may permit or require grouping in this situation to be specified by a list of grouping attributes. In this case, the rows are grouped prior to aggregation such that each group corresponds to each distinct combination of values for all of the specified grouping attributes.

Some query languages permit query blocks to be nested such that the predicate of an "outer" query block includes reference to the result of an "inner" query block. In this case, the query language specifies a result that would be obtained by re-evaluating the inner query block each time the outer query block evaluates the predicate for a different row or combination of rows when performing its specified restriction or join operations. Evaluating the result in such a fashion, however, usually is very inefficient.

To select the most efficient of alternative ways of evaluating a query, the database management system typically includes a query optimizer. In general, a query can be evaluated a number of ways, because, in many cases, the query operations obey certain commutative, associative, or distributive laws. By applying these laws, the query optimizer may formulate alternative orders of performing the query operations, compute a cost of performing each such "query plan", and select the least costly query plan for execution.

Unfortunately, if a query is specified by nested query blocks, the nesting itself specifies an iterative order of performing query operations. Since the query operations of an inner block are specified to occur during the restriction or join operation of the outer query block, the query optimizer cannot apply the laws permitting alternative execution orders unless the nested query blocks are first "un-nested" into equivalent "pipelined" query blocks.

When query blocks are pipelined, the result of a first query block is specified as input to the predicate of a second query block, but that result is presumed to be evaluated only once before evaluation of the second query block. Therefore, there is a distinct order of performance specified between the first query block and the second query block. In many cases the un-nesting process generates un-nested query blocks in which this specified order is more efficient that the iterative method of executing the original nested query blocks. In other cases, the optimizer can find an even better order of execution for evaluating the un-nested query blocks.

The problem of un-nesting and optimizing database queries has been studied extensively. A general solution was proposed in Won Kim, "On Optimizing an SQL-like Nested Query", ACM Transactions on Database Systems, Vol. 9, No. 3, American Association for Computing Machinery, United States (1982), incorporated herein by reference. Later, it was discovered that the un-nesting technique of Kim does not always yield the correct results for nested queries that have non equi-join correlation predicates or that have a "COUNT" aggregate between the nested blocks. Un-nesting solutions for these anomalous cases were described in Richard A. Ganski and Harry K. T. Wong, "Optimization of Nested SQL Queries Revisited", Proceedings of the Sigmod Conference, American Association for Computing Machinery, United States (1987), pp. 23-33, incorporated herein by reference. Ganski's method of

un-nesting was extended to multiple nested blocks as disclosed in Umeshwar Dayal, "0f Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates and Quantifiers", Proceedings of the 13 VLDB Conference, Brighton, 1987, pp. 197–208, incorporated herein by reference. Methods of un-nesting multiple blocks within the same block are disclosed in M. Muralikrishna, "Optimization and Dataflow Algorithms for Nested Tree Queries", Proc. VLDB Conf. (August 1989), pp. 77–85, incorporated herein by reference. The commuting of joins and outer joins under specific conditions is disclosed in Arnon Rosenthal and Cesar Galindo-Legaria, "Query Graphs, Implementing Trees, and Freely-Reorderable Outerjoins", Proc. SIGMOD Conf (May 1990), pp. 291–299, incorporated herein by reference.

SUMMARY OF THE INVENTION

Briefly, in accordance with the present invention, there is provided an alternative method of un-nesting queries with equi-join predicates and a "COUNT" aggregate between the nested blocks. This alternative is used where it is more efficient, than the method of Ganski, supra. The method of the present invention extends the semantics of the outer join operator to permit the application of different predicates to the join tuples and the anti-join tuples. The anti-join tuples, for example, are associated with a count value of zero.

A further aspect of the present invention involves the use of the novel un-nesting method to queries with multiple blocks. In this case, the alternative un-nesting method of the present invention is used when the correlation predicates in the Count block are "neighbor predicate" referencing the relation in their own block and the relation from the immediately enclosing block. Otherwise, the Ganski-Dayal un-nesting method is used.

Still another aspect of the present invention is an integrated procedure for evaluating joins and outerjoins in a top-down order. This integrated procedure enables the alternative un-nesting method of the present invention to be applied to more blocks.

BRIEF DESCRIPTION OF THE DRAWINGS

Other objects and advantages of the invention will become apparent upon reading the following detailed description and upon reference to the drawings in which:

FIG. **1** is a block diagram of a database management system incorporating the present invention;

FIG. **2** is a block diagram of a relational database used in the database management system of FIG. **1**;

FIG. **3** is a flowchart of the basic steps followed by the database management system of FIG. **1** for processing a query;

FIG. **4** is a block diagram of a query node generated during the parsing of a query block by the database management system;

FIG. **5** is a flowchart of a procedure followed by the database management system of FIG. **1** when parsing a query block;

FIG. **6** is a flowchart of a procedure used by the database management system when parsing a predicate in a query block;

FIG. **7** is a flowchart of a procedure for executing a query block;

FIG. **8** is a detailed flowchart of a specific implementation of the flowchart of FIG. **7** in accordance with tuple iteration semantics;

FIG. **9** is a data flow graph representation of two nested query blocks;

FIG. **10** is a data flow representation of two pipelined query blocks obtained by applying Kim's un-nesting method to the nested query blocks of FIG. **9**;

FIG. **11** is a data flow representation of two pipelined data blocks obtained for a count aggregate when the nested query blocks of FIG. **9** are un-nested by Ganski's method;

FIG. **12** Is a data flow representation of two pipelined query blocks obtained for a count aggregate when the nested query blocks of FIG. **9** are un-nested by the method of the present invention;

FIG. **13** is a flowchart of a specific procedure for performing a left outer join operation upon a count aggregate and applying different predicates to the joining and anti-joining tuples;

FIG. **14** is a join graph used by the query optimizer of the present invention;

FIGS. **15** and **16** together comprise a flowchart of an integrated un-nesting procedure in accordance with the present invention;

FIGS. **17** to **22** are six alternative query graphs obtained by un-nesting the query graph shown in FIG. **14** in accordance with the procedure of FIGS. **15** and **16**.

While the invention is susceptible to various modifications and alternative forms, a specific embodiment thereof has been shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that it is not intended to limit the invention to the particular form disclosed, but, on the contrary, the intention is to cover all modifications, equivalents, and alternatives falling within the spirit and scope of the invention as defined by the appended claims.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

Turning now to FIG. **1** of the drawings, there is shown a database management system generally designated **30**. The database management system **30** is made by programming a conventional digital computer. The database management system **30** includes a memory **31** storing a relational data-base **32**, as will be further described below with reference to FIG. **2**. For processing data in the relational data-base **32**, the database management system **30** includes a data processor **33**. For communicating with a user **34**, the database management system includes an input device **35**, such as a keyboard, and an output device **36**, such as a video display.

The present invention more particularly concerns the processing of a query from the user **34** instructing the database management system **30** to display to the user specified data from the relational database **32**. The query is received by the input device **35** and processed by a query processing system **37** in the data processor **33**. In particular, the query processing system is made by loading a computer program into the data processor **33** and executing that computer program. The query processing system **37** includes a parser **38** which converts the query from the user **34** into an internal representation that identifies and locates specific components of the query. The internal representation, for example, is a hierarchical graph constituting one of the query graphs **39** stored in the memory **31**.

The query graph of the parsed query specifies a particular order of execution, but the specified order of execution is not necessarily the optimum way of execut-

ing the query. Instead, there may be any number of better alternative ways of executing the query to obtain identical results. To possibly obtain a more optimum execution plan, the query processing system **37** includes a query optimizer **40** that operates upon the query graph of the parsed query to generate a number of alternative query plans, each specified by a respective one of the query graphs **39**. The query optimizer **40** computes a cost associated with each of the alternative query plans, and executes the query plan having the least cost. During execution of this query plan, a retrieval system **41** accessing data from the relation database **32**, and performs upon the data a number of relational operations. These relational operations result in an output relation which is transmitted to the output device **36** and displayed to the user **34**.

Turning now to FIG. **2**, there is shown a detailed representation of the relational database **32**. The relational database **32** includes data stored in a number of tables, such as the tables **51** and **52**. Each table corresponds to a particular relation in the relational database. The table **51**, for example, corresponds to a relation "R" and the table **52** corresponds to a relation "S". Each table is shown as a rectangular matrix of rows and columns. Internally, however, the data could be stored in any number of ways. Each row, for example, could be stored as a record of a file. Alternatively, the data in each of the tables could be freely dispersed in memory, and the rows and columns could be defined by lists of pointers to the data. In any event, the data structures storing the data in the tables **51** and **52** permit all of the rows in a table to be accessed sequentially and further permit a specified column position of an accessed row to be indexed.

Each column in each of the tables is indexed by a key called an attribute. Most conventional database query languages assign an alphanumeric relation name to each table and further assign an alphanumeric attribute name to each column of each table, but the attribute name associated with one table need not be distinct from the attribute name of another table. A particular column of data, however, can be uniquely specified by the combination or concatenation of both a relation name and an attribute name.

To index the columns of the tables, the relational database **32** further includes a number of indices, or definitions of keys, to the tables and columns. These indices are known as a schema **53** corresponding to a particular subject domain. The schema **53** is further subdivided into an index for each relation, such as an index **54** to the relation "R" of the first table **51**, and an index **55** to the relation "S" for the second table **52**. The index **54** to the relation "R" includes a name ("R") for the relation and a name ("a", "b", "c", "d", "e", "f", "g") for each attribute corresponding to each column of the table R, and the index **55** to the relation "S" includes a name ("S") for the relation and a name ("h", "i", "j", "k", "l") for each attribute corresponding to each column of the table **52**. The indices **54** and **55**, for example, permit a numerical pointer or address to be obtained to a table and column specified by a relation name and an attribute name.

The indices **54** and **55** may further include respective indices **56** and **57** to the individual rows of the tables. For processing most relational operations, it is only necessary to scan sequentially through all of the rows of a table. To facilitate the selection and retrieval of a particular row, however, it is desirable to identify one or more columns of the table as including keys to the individual rows. Some operating systems, for example, have optimized procedures for storing tables in files according to a primary key and for retrieving a row or record specified by the primary key. In such systems, if the user does not specify a column as including the primary key, the system may nevertheless assign a primary key to each row. In such a case, the primary key is known as a "surrogate column" which, if not accessible to the user, will be accessible to the computer programmer.

Turning now to FIG. **3**, there is shown a flowchart of the basic procedure followed by the query processing system (**37** in FIG. **1**) when processing a query. In a first step **61**, a root node is cleared for a query graph to receive a query from the user **34**. Next, in step **62**, a current node pointer is set to point to the root node. Then, in step **63**, a query is received from the user **34**.

The query is a string of alphanumeric characters which conform to a query language that identifies data in the relational database and operations to perform upon the data. In many conventional query languages, the query is in the form of a query block, which may refer to additional nested query blocks, as will be further described below.

To process the query, the query is converted or parsed from the variable-length format of the query language into a more fixed format for processing by the query optimizer (**40** in FIG. **1**) and execution by the retrieval system (**41** in FIG. **1**). The query block is parsed in step **64**, for example, by calling a subroutine further described below with reference to FIG. **5**. Parsing of the query block generates a node in the query graph. If the parsed query block includes nested nodes, the nested nodes are parsed recursively, creating additional nodes in the query graph that are linked to the root node. Once the query block is parsed, execution continues from step **64** to step **65**.

If the query from the user fails to conform to the syntax of the query language, the parsing in step **64** will return with an error. If so, then further processing of the query cannot continue, and execution therefore returns, so that the user may submit a corrected query. Otherwise, execution continues in step **66** by optimizing the query.

Query optimization is further described below with reference to FIGS. **9** to **22**. The query optimization results in a number of alternative query plans represented by respective query graphs (**39** in FIG. **1**). The optimizer (**40** in FIG. **1**) selects the query plan having the least computational cost, and, in step **67**, that query plan is executed. The query plans includes nodes, each of which corresponds to the operations of a single query block. The execution of a single query block is further described below with reference to FIGS. **7**, **8** and **13**.

For the sake of illustration, the syntax for a specific query language will now be described. This syntax will roughly correspond to "Structured Query Language" (SQL) as adopted by the American National Standards Institute in 1986. The use of such a standard query language in the context of a relational database system is described in Hobbs and England, *Rdb/VMS—A Comprehensive Guide*, Digital Equipment Corporation, Maynard, Mass. (1991), incorporated by reference, on pages 36–64.

It is possible to represent virtually any query by a pipelined series of query blocks, each of which specifies a list of relations, a predicate for restricting or joining

rows of the relations, a list of columns to be selected as output, and a list of aggregate functions. If at least one column and at least one aggregate function is specified, it is also desirable to specify groups of rows over which the aggregate functions are to be performed. Typically, it is desirable to group rows having identical values in specified columns. Therefore, regardless of the particular syntax of the query language, virtually any query to a relational database can be represented as a graph of nodes having a format similar to the format shown in FIG. 4.

As shown in FIG. 4, a query node **80** includes a pointer **81** to its parent node in the query graph, an attribute list **82** specifying the columns in an output relation, a "GROUPBY" list **83** specifying the columns for grouping rows having identical values in the columns, an aggregate list **84** of the aggregate functions to perform upon the grouped rows, a relation list **85** specifying the input to the query node, a predicate **86** to be tested against combinations of rows of the input relations, pointers **87** to nested children of the query node **80**, and pointers **88** to pipelined children of the query node **80**.

The hierarchical structure of a query graph including the query node **80** specifies an order of execution among the query nodes. According to the specified order, the root node of the query graph is first inspected to determine whether it has any pointers to pipelined children. If so, then the pipelined children nodes are inspected in a recursive fashion until a node is found having no pipelined children. This so-called "leaf" node in the query graph is executed, which may require the execution of nested query blocks specified by the pointers to the nested children. Once execution of a query node is completed, execution returns to its parent node. In this way, the specified order of execution is from the bottom of the query graph to the top of the query graph. Once the root query node is executed, the resulting relation is transmitted as output to the user.

The execution order specified by the query graph might not be the best order of execution. Therefore, when a query block is parsed, the parser (**38** in FIG. **1**) may identify node-type information **89** for use by the query optimizer (**40** in FIG. **1**). The node-type information **89**, for example, specifies whether the node has an aggregate function, whether the aggregate function is a count function, whether the predicate is an equi-join predicate, whether the node is a nested node of a parent node, and whether a nested node references a relation in a parent or ancestor node in which it is nested.

For describing the parser **38**, it is necessary to assume a specific format for a query from the user. A query will include, for example, a "SELECT" statement, a "FROM" statement, possibly a "WHERE" statement, and possibly a "GROUPBY" statement, arranged as follows:

SELECT [Attribute list] [Aggregate list]
FROM [Relation list]
[WHERE [Predicate]]
[GROUPBY [Attribute list]]

When there is a "WHERE" statement having a predicate, the predicate should have two terms joined by a relational operator, and possibly one or more logical operators joining similar combinations of terms as follows:

[term] [Relation Operator] [term]

-continued

[Logical operator] [term] [Relational operator] [term]
[Logical operator] [term] [Relational operator] [term]
. . .

Turning now to FIG. **5**, there is shown a flowchart of a procedure for parsing a query block having the above format. In a first step **91**, a first word in the query is parsed. If the first word is not "SELECT", as tested in step **92**, then a syntax error has occurred. The user is notified of the error in step **93**, and execution returns. Otherwise, in step **94**, the next word in the query is parsed. If the word is not "FROM", as tested in step **95**, then the word should be part of the attribute list or the aggregate list. The word is compared to reserved words for predetermined aggregate functions (such as COUNT or SUM), and if the word is recognized as an aggregate, it is placed in the aggregate list. Otherwise, the word is placed in the attribute list. If the word does not correspond to an alphanumeric name for an attribute, then an error has occurred and execution returns. Otherwise, execution loops from step **96** back to step **94** to parse the next word in the query. Eventually, step **95** will recognize the word "FROM" beginning the relation list. In step **97**, the next word is parsed, and in step **98**, the word is compared to "WHERE". If the word is not "WHERE", then it should be a relation. Therefore, execution branches to step **99**. In step **99**, execution branches to step **103** if the word is "GROUPBY". Otherwise, execution continues to step **100**. In step **100**, execution returns if the end of the query is reached. If not, then execution continues to step **101** to add the word to the relation list of the query block. If the word does not correspond to the name of a predefined relation, however, an error has occurred, and execution returns. Otherwise, execution loops back to step **97**.

If step **98** finds that the word is "WHERE", then execution continues to step **102** to parse the predicate of the "WHERE" statement. The predicate is parsed as described below in FIG. **6**. After parsing the predicate, execution continues in step **103** to determine whether a "GROUPBY" statement follows the "WHERE" statement. If not, the end of the query block has been reached and execution returns. Otherwise, in step **103**, the following words are parsed into the "GROUPBY" list.

The procedure of FIG. **5** is also used when parsing an inner nested query block. In this case, a step **105** is executed which allocates a child node for the nested query block, and links the child node to the parent node in the query graph. Step **105** is called when the predicate of an outer query block is parsed and the word "SELECT" is discovered as a term in the predicate, as further described below with reference to FIG. **6**. After step **105**, execution continues in step **94** to parse the next word in the inner nested query block.

Turning now to FIG. **6**, there is shown a flowchart of a procedure for parsing the predicate. In a first step **116**, a first term of the predicate is parsed. The term should either be a constant, a predefined attribute, the end of the query block, the beginning of a GROUPBY statement, or the word "SELECT" specifying a nested query block. If the end of the query block is reached, as tested in step **117**, execution returns. If the term is GROUPBY, as tested in step **118**, execution also returns If the term is "SELECT" as tested in **119**, then in step **120**, the procedure of FIG. **5** is called recursively to

parse the nested query block. In any case, execution continues in step **121** to parse a comparison operator. If a predefined comparison operator is not found, then an error has occurred and execution returns. Otherwise, in step **122**, a next term is parsed. If the term is "SELECT", as tested in step **123**, then the term is a nested query block, and execution branches to step **124** to recursively call the procedure of FIG. **5** to parse the nested query block. In any case, execution continues in step **125** to compare the type of the two previous parsed terms to the type required by the comparison operator parsed in step **121**. If the type of the terms do not agree with the comparison operator, then an error has occurred and execution returns. Otherwise, execution continues to step **126**. Step **126** attempts to parse a logical operator. The predicate, however, might not have a logical operator. Step **127** determines whether the end of the query block has been reached. If so, execution returns. Otherwise, step **128** determines whether the beginning of a "GROUPBY" statement has been reached. If so, execution returns. Otherwise, assuming that a valid logical operator has been parsed, execution continues to step **129**, where a next term is parsed. Execution then loops back to step **119**. Therefore, additional terms and comparison operators are parsed until the end of the predicate is reached in step **127** or **128**.

Specific programming for parsing expressions and evaluating expressions by symbolic execution is found in Hardy et al., U.S. Pat. No. 4,648,044, issued Mar. 3, 1987, incorporated herein by reference.

Turning now to FIG. **7**, there is shown a flowchart of a procedure for executing a query block. This procedure follows the specified way of executing a SQL query block having a "SELECT", "FROM", "WHERE" and possibly a "GROUPBY" statement. This particular procedure should be modified, for example as shown in FIG. **13**, to perform the method of the present invention.

In a first step **131** of FIG. **7**, a new relation is formed having rows which are all combinations of all rows of all relations in the relations list. Then, in step **132**, the predicate is tested for each row of the new relation, and rows are discarded for which the predicate is false. In other words, step **132** performs a restriction operation upon the new relation formed in step **131**. In step **133**, execution branches depending upon whether the aggregate list is empty. If there are not any aggregates specified by the aggregate list, then in step **134**, a projection operation is performed upon the remaining rows of new relation by selecting the columns specified by the attribute list, and execution returns.

If step **133** determines that aggregates are specified, then execution branches to step **135**. In step **135**, execution branches depending on whether the "GROUPBY" list is empty. If the "GROUPBY" list is empty, then in step **136**, an aggregation is performed over all remaining rows of the new relation. Then in step **137**, the aggregates are selected for output, and execution returns.

If step **135** finds that the GROUPBY list is not empty, then in step **138**, an aggregate is performed over groups of rows having the same values in all of the columns specified by the attribute names in the "GROUPBY" list. Then, in step **139**, the columns specified by the attribute list are selected for output, together with the aggregates, and execution returns.

FIG. **7** defines the conceptual operations required for executing the query block. In practice, however, these conceptual operations can be performed in various ways. A fairly straightforward implementation is known as "tuple iteration" because it executes the query block by sequentially indexing the rows of the relations in the relation list so that all combinations of rows are indexed.

Turning now to FIG. **8**, there is shown a flowchart of a procedure for evaluating a query block by tuple iteration. In a first step **151**, a row pointer P_i is allocated to each relation R_i in the relation list. Then, in step **152**, all of the row pointers are set to zero, and a variable NRO, which is used to indicate the number of output rows, is also set to zero. At this point, iteration may begin. In step **153**, execution branches depending on whether the predicate is null. If the predicate is null, then execution branches to step **154**. In step **154**, execution branches depending on whether there is an empty aggregate list. If the aggregate list is empty, then in step **155**, a row of the output relation is formed with the columns specified by the attribute names in the attribute list. This is the end of the operations for a single iteration or combination of rows from the relations in the relation list. Since this iteration generated a new output row, the variable NRO is incremented by one in step **155**.

Execution continues in step **157** to begin a series of tests which determine the next combination of row pointer values to index the next combination of input rows. In step **157**, an index I is set to zero. Step **158** then checks whether the Ith row pointer has reached its maximum value of PMX(I). If so, then in step **159**, this pointer is set to zero. Then, in step **160**, the index I is compared to NR, which is one less than the number of relations in the relation list. Step **160** checks whether the index I has reached its maximum value. If so, then no further iteration is necessary. In step **161**, the columns of output relation which are specified by the attribute list are selected for output together with any aggregates, and execution returns. Otherwise, in step **162**, the index I is incremented, and execution loops back to step **158** to check the next row pointer. Eventually, a row pointer will be found in step **158** to be less than its maximum, or else all pointers will be at their maximum values and execution will return from step **160**. If the Ith row pointer is found in step **158** to be less than its maximum value, then in step **163**, that row pointer is incremented by one and execution loops back to step **153** to begin another iteration.

When step **153** determines that the predicate is not null, then in step **171**, the predicate is tested for values from the rows $R_i(P_i)$. In the next step **172**, execution branches depending upon whether the predicate is true. If so, execution branches to step **154** to further process that row. Otherwise, execution continues in step **157**.

When step **154** determines that the aggregate list is not empty, then execution branches to step **173**. In step **173**, execution branches depending on whether the "GROUPBY" list is empty. If the "GROUPBY" list is empty, then the input row (P_i) is aggregated. The columns over which the input row is aggregated are specified by attribute arguments of the aggregate functions. The result is a single aggregate row, and therefore in step **175**, the variable NRO is set to one. Execution then continues in step **157**.

If step **173** determines that the "GROUPBY" list is not empty, then execution branches to step **176** to determine whether the input row (P_i) is part of a new group, or whether it is part of an already existing output group. This determination is made by scanning all of the existing output rows. In step **176**, an output row pointer RO

is set to zero. Then, in step **177**, the output row pointer RO is compared to the number of output rows NRO. If the row pointer RO is equal to the number of output rows, then execution branches to step **178**. In step **178**, the input row (P_i) is aggregated with a new output row (RO). Execution then continues in step **155** to increase the number of output rows.

If, in step **177**, it was found that the row pointer RO has not yet reached the current number of output rows NRO, then in step **179**, the rows (P_i) of the input relation are compared to the rows (RO) of the output relation in all of the columns specified by the attribute list. In this regard, the aggregation process forms a new relation that includes the columns specified by the attribute list, any columns specified by the "GROUPBY" list not included in the attribute list, and a column for each aggregate function. Any column specified by the "GROUPBY" list that is not included in the aggregate list is a temporary column used by the scanning procedure, but which is not selected for output in step **161** at the end of evaluation. In the next step **180**, execution branches to step **181** if there is not a match. In step **181**, the scan index RO is incremented and execution loops back through steps **177**, **179**, and **180** until either a match is found in step **180** or all of the rows are scanned, as tested in step **177**.

If step **180** finds a match, then in step **182**, the input rows (P_i) are aggregated with the output row (RO) and execution then continues in step **157**.

The present invention more particularly concerns un-nesting queries. Consider, for example, the following two-block query:

EXAMPLE 1

```
SELECT R.a
FROM R
WHERE R.b OP1 (SELECT COUNT (*)
        FROM S
        WHERE R.c = S.c)
```

In this and the following examples, relations are named by capital letters, and attributes are named by small letters. Attribute names will be fully qualified by their relation names. Therefore, "R.c" represents the attribute c of the relation R.

A query graph of this query is shown in FIG. **9**. A node **191** represents the operations of the inner query, and a node **192** represents the operations of the outer query. In Example 1 above, the predicate (R.c=S.c) of the inner query references the relation R in the outer query. Therefore, the inner node cannot be precomputed.

The un-nesting procedure of Kim, supra., transforms the above query into the following two un-nested queries:

Query 1:
TEMP (c, count)=
SELECT S.c, COUNT (*)
FROM S
GROUPBY S.c

Query 2:
SELECT R.a
FROM R, TEMP
WHERE R.c=TEMP.c and R.b OP_1 TEMP.count

The result of the first query is a temporary relation TEMP that is pipelined into the second query. The first query computes a count value associated with every distinct value in the c attribute of S.

A graph of the un-nested queries is shown in FIG. **10**. The first query is represented by a node **197**, and the second query is represented by a node **198**. The first query computes a count for each distinct value of S.c. Kim concluded that the second query of FIG. **10** would give the same result as the outer query of FIG. **9** because the join predicate "R.c=TEMP.c" of node **194** would select the same aggregate for each value of R.c as would be computed by the inner node **191** of FIG. **9**. Unfortunately, this conclusion incorrectly presumes that there is some value of S.c associated with each count computed by the inner query of FIG. **9** for each row of R. Therefore, Kim's method of FIG. **10** will not work correctly for a count aggregate as shown. Kim's method, however, will work correctly for other aggregates such as MIN, MAX, SUM and AVERAGE.

Consider now the following specific example where the aggregate is the count function:

EXAMPLE 2

```
SELECT DEPT.id
FROM DEPT
WHERE DEPT.pcs > (SELECT COUNT (*)
        FROM EMP
        WHERE EMP.dept_name = DEPT.name)
```

This nested query would find all departments that have more work stations than employees. The "*" in the argument list of the COUNT function denotes a count of all combinations of tuples that satisfy the predicate "EMP.dept_name= DEPT.name". If an attribute occurs as an argument of the COUNT function, then a count is made of all such tuples that have a non-null value for the specified attribute.

Applying Kim's un-nesting method to the nested query of Example 2 produces the two following queries:

TEMP (dept_name, count)=
SELECT EMP.dept_name, COUNT (*)
FROM EMP
GROUPBY EMP.dept_name
SELECT DEPT.id
FROM DEPT, TEMP
WHERE DEPT.name=TEMP.dept_name
AND DEPT.pcs>TEMP.count

If there is a new department with no employees, but at least one work station, then Kim's method gives an incorrect result. In this case, the relation DEPT will include a tuple or row with DEPT.pcs>0, but the relation EMP will not include any tuple or row "associated" with that tuple or row. In more precise terms, the tuple with DEPT.pc>0 of DEPT.pcs will not have any tuples in EMP that will be joined with it by the join correlation predicate DEPT.name=EMP.dept_name. Therefore, the WHERE statement of the original query will have its predicate satisfied by that tuple because DEPT.pcs will have a value>0 and COUNT (*) will have a value of 0. But the un-nested query will not have its predicate satisfied by that tuple because the join correlation predicate DEPT.name=TEMP.dept_name will not be true for that particular department.

A solution to the so-called "count bug" was described in *Ganski*, supra. With respect to Example 1 above, for the case of AGG being the COUNT function, Ganski reasoned that a tuple r of R would be lost

after the join in Kim's second query if it does not join with any tuples of S. However, the COUNT associated with r is 0 and if R.b OP_1 0 is true, tuple r should appear in the result. In order to preserve tuples in R that have no joining tuples in S, Ganski proposed that a (left) outer join (OJ) should be performed when the COUNT aggregate is present between two blocks. In this case, the un-nested query becomes:

```
Query 1:
TEMP (c, count) =
SELECT R.c, COUNT (S.c)
FROM R.S
WHERE R.c OJ S.c
GROUPBY R.c
Query 2:
SELECT R.a
FROM R, TEMP
WHERE R.c = TEMP.C
      AND R.b OP1 TEMP.count
```

A corresponding query graph is shown in FIG. **11**. The first query is represented by a node **195**, and the second query is represented by a note **196**. The (left) outer join operation (OJ) preserves every tuple of R, and consequently a count is computed for every distinct value of R.c. Consider, for example, the following tuples for the relations R and S:

R.a	R.b	R.c	S.c
Car	1	Red	Red
Truck	2	Green	Blue
Boat	4	Blue	Blue
House	0	Orange	Black

A join with a join correlation predicate of R.c=S.c would create a new relation with the following tuples:

R.a	R.b	R.c	S.c
Car	1	Red	Red
Boat	4	Blue	Blue
Boat	4	Blue	Blue

If the tuples were grouped by distinct joining values of R.c or S.c and aggregated by COUNT(S.c), the following relation TEMP would be generated:

R.c	S.c	COUNT (S.c)
Red	Red	1
Blue	Blue	2

The counts for this relation would be different from the counts for distinct values of R.c generated by iterative evaluation of the inner query in FIG. **9** because it would never return a count of zero. By replacing the join with a left outer join, a new relation with the following tuples would be generated:

R.a	R.b	R.c	S.c
Car	1	Red	Red
Truck	2	Green	^
Boat	4	Blue	Blue
Boat	4	Blue	Blue

-continued

R.a	R.b	R.c	S.c
House	0	Orange	^

For every tuple of R having a value for R.c which does not join with a value of S.c for any tuple of S, a new tuple is generated by appending null values ( ) to that tuple of R. The tuples of R having a value for R.c which does not join with a value of S.c for any tuple of S will be referred to as the anti-joining tuples of R. Because a count over a specified attribute will not count null values for that attribute, a count over the column S.c of the outer join will return a value of zero for the anti-join tuples:

TEMPOJ (c, count) =

R.c	COUNT (S.c)
Red	1
Green	0
Blue	2
Orange	0

Ganski's method avoids the count bug by providing a correct count associated with each distinct value of R.c. Ganski's method also provide proper un-nesting where the inner query does not have an equi-join correlation predicate. An equi-join correlation predicate is of the form f1(R)=f2(S) where f1 and f2 are any functions on tuples of R and S, respectively.

By applying Ganski's method to the nested query blocks of Example 2, the following un-nested queries are obtained:

```
TEMP (name, count) =
SELECT DEPT.name, COUNT (EMP.dept_name)
FROM DEPT, EMP
WHERE DEPT.name OJ EMP.dept_name
GROUPBY DEPT.name
SELECT DEPT.id
FROM DEPT, TEMP
WHERE DEPT.name = TEMP.name
      AND DEPT.pcs > TEMP.count
```

Ganski's method can convert two nested query blocks into a single query in SQL query language. In this case, the operation of the second query is performed by a "HAVING" statement. A GROUPBY statement specifies grouping according to a primary key or surrogate column having a distinct value for each tuple or row. Such an attribute will be designated by "#". The result for Example 2 is as follows:

```
SELECT DEPT.id
FROM DEPT, EMP
WHERE DEPT.name OJ EMP.dept_name
GROUPBY DEPT.#
HAVING DEPT.pcs>COUNT (EMP.dept_name)
```

In accordance with the present invention, there is provided an alternative method of un-nesting a nested query having a count aggregate. This alternative method is illustrated by the query graph in FIG. **12**. The first query, represented by the node **198**, is similar to the first query of Kim's method (node **193** in FIG. **10**). In the second query, represented by node **197** of FIG. **12**, the method of the present invention performs a (left) outer join, and applies a different predicate to the anti-joining tuples than to the joining tuples. In particular,

when a nested inner query has an equi-join predicate joining a relation of the inner query to an outer query and a count aggregate, the method of FIG. **12** removes the equi-join predicate from the inner query and places a corresponding conjunctive (left) outerjoin predicate term in the predicate of the outer query, performs the count aggregate for each distinct value of the joining attribute of the relation of the inner query, and in the outer query applies different predicates to the joining and anti-joining tuples such that the predicate of the anti-joining tuples is evaluated assuming a count value of zero.

In Example I, for a count aggregate, the method of FIG. **12** gives the following two un-nested queries:

```
TEMP (c, count) =
SELECT S.c, COUNT (*)
FROM S
GROUPBY S.c
SELECT R.a
FROM R, TEMP
WHERE R.c OJ TEMP.c
    AND ( IF R.c =TEMP.c
      THEN R.b OP1 TEMP.count
      ELSE R.b OP1 0)
```

As will be described below with respect to FIG. **13**, the IF-THEN-ELSE operations are best performed by directly applying the predicate operation OP_1 between the left join relation R and a count value when a joining tuple of R is found, and applying the predicate operation OP_1 between the left join relation R and a value of 0 when an anti-joining tuple of R is found. In the following examples, the result of the IF-THEN-ELSE operations will be expressed simply as first the predicate applied to the join tuples and second the predicate applied to the anti-join tuples as follows:

[R.b OP_1 TEMP.count|R.b OP_1 0]. With this notation, the query of Example 2 is un-nested as follows:

```
TEMP (dept_name, count) =
SELECT EMP.dept_name, COUNT (*)
FROM EMP
GROUPBY EMP.dept_name
SELECT DEPT.id
FROM DEPT, TEMP
WHERE DEPT.name OJ TEMP dept_name
    AND [DEPT.pcs > TEMP.count ¦ DEPT.pcs > 0]
```

Under certain circumstances, the method of FIG. **12** may be more efficient than Ganski's method. The heuristic argument is based on (1) the number of tuples that flow from each node in the query graphs corresponding to the two methods, and (2) the number of tuples that have to be processed by each groupby and outer join operation. Both methods involve accessing relations R and S. Clearly $|TEMP_1| \leqq |S|$ and $|R| \leqq |R \text{ OJ } S|$. Assume that $|S| < |R|$. The number of tuples flowing from the groupby operation to the outer join operation in the method of FIG. **12** is equal to $|TEMP_1|$. The number of tuples flowing from the outer join operation to the groupby operation in Ganski's method is equal to $|R \text{ OJ } S|$. Clearly $|TEMP_1| < |R \text{ OJ } S|$. The number of tuples processed by the groupby operation and the outer join operation in the method of FIG. **12** is each less than the corresponding number of tuples in Ganski's method. Hence, if $|S| < |R|$, the method of FIG. **12** should perform better than Ganski's method. This heuristic argument, however, ignores the fact that Ganski's method joins two base relations, whereas the method of FIG. **12** joins a base relation with a temporary relation. As a result, Ganski's method might be able to employ more join methods, and the optimizer should take any alternative join methods into account. In any event, the present invention provides a different way of un-nesting a nested query having a COUNT aggregate when the correlation predicates are all equi-joins.

Turning now to FIG. **13**, there is shown a flowchart of a procedure for performing a left outer join and applying a predicate to the join tuples different from the predicate applied to the anti-join tuples. In particular, the count associated with the anti-join tuples is modified so that it has a value of zero for the anti-join tuples. The procedure of FIG. **13** takes advantage of the fact that the relation TEMP has a column with an attribute c that has distinct values. The row pointer P_j to the relation TEMP is used as an index in an inner loop, and if no matches to the distinct column c occur, then an anti-join tuple is formed with the relation R. Otherwise, if a joining tuple is found, the inner loop is exited since the join is performed with a distinct value of TEMP.c, and therefore no other joining tuple would be found by further iterations.

In a first step **221**, a row pointer P_j is allocated to the relation TEMP, and a row pointer P_k is allocated to the relation R. Then, in step **222**, the row pointers are set to zero. Both the inner and outer loops begin in step **223** with the join comparison between the row P_k and attribute c of the relation R and the row P_j and attribute c of the relation TEMP. If joining tuples are found, then the inner loop is exited to step **224**. In step **224**, the predicate for the joining tuples is applied. In this case, a test is made of the operation OP_1 between the attribute R.b at row P_k and the attribute TEMP.count at row P_j. If this predicate is found to be true in step **225**, then in step **226**, an output row is formed by joining the P_kth row of the relation R with the P_jth row of the relation TEMP.

Returning now to step **223**, if joining tuples are not found in step **223**, then in step **227**, the end of the inner loop is tested. If the row pointer P_j is not equal to its maximum value PMXj, then the inner loop continues in step **228** by incrementing the row pointer P_j and execution loops back to step **223**. If the row pointer P_j reaches its maximum value in step **224**, then there are no joining tuples for the value in the P_kth row of the attribute R.c. Therefore, an anti-join tuple pair has been found. In this case, an anti-join predicate is tested in step **229** that is different from the join predicate applied in step **224**. For the anti-join predicate **229**, it is assumed that the value of the attribute TEMP.count has a value of zero. If this anti-join predicate is true, as tested in step **230**, then in step **231**, an output row is formed by joining the P_kth row of the relation R with a value of the P_kth row of the attribute R.c, and the value of zero. In other words, the row **231** is the row that would have been present in the absence of the so-called "count bug".

From steps **231** and **226**, or if a false logical value is found in steps **225** or **230**, then in step **232**, the end of the outer loop is tested by comparing the pointer P_k to a predetermined maximum value PMX_k. If this maximum value is reached, execution returns. Otherwise, execution continues in step **233** by setting the row pointer P_j to zero, and incrementing the row pointer P_k by one. Execution then loops back to step **223**.

The method of FIG. **4** is readily applicable to nested queries of more than two levels of nesting so long as the

correlation predicates of the COUNT blocks are "neighbor predicates". By definition, a "neighbor predicate" is an equi-join correlation predicate that references the relation in its own block and the relation from the immediately enclosing block. If a COUNT block does not have a "neighbor predicate", then the method of FIG. 12 is more difficult to apply than Ganski's method of FIG. 11, and consequently Ganski's method should be used for un-nesting of the count block. The use of the method of FIG. 12 for un-nesting blocks having more than two levels of nesting is shown in the following example:

EXAMPLE 4

```
SELECT R.a
FROM R
WHERE R.b < (SELECT COUNT (*)
      FROM S
      WHERE R.c = S.c
      AND S.d > = (SELECT COUNT (*)
            FROM T
            WHERE S.e = T.e))
```

Notice that all correlation predicates are neighbor predicates.

Kim, supra, page 465, disclosed that his method is applicable to multiple nested blocks by applying it to pairs of inner and outer nested blocks beginning at the bottom of the hierarchical query graph. In other words, a search is made down to a leaf node of the query graph, and the method for un-nesting of a pair of blocks is applied recursively to an inner nested leaf node and its outer parent node. *Ganski*, supra, pages 31–32, proposed a similar recursive procedure upon the query graph to apply his method upon multiple nested blocks. The same approach can be used to apply the method of FIG. 12. The result of the query is obtained by evaluating the following three queries.

```
TEMP1 (e, count) =
SELECT T.e, COUNT (*)
FROM T
GROUPBY T.e
TEMP2 (c, count) =
SELECT S.c, COUNT (*)
FROM S, TEMP1
WHERE S.e OJ TEMP1.e
    AND [S.d > TEMP1.count ¦ S.d > = 0]
GROUPBY S.c
SELECT R.a
FROM R, TEMP2
WHERE R.c OJ TEMP2.c
    AND [R.b < TEMP2.count ¦ R.b < 0]
```

Although recursive operations upon query graphs are both a convenient and readily understood way of extending a method for un-nesting two nested queries to nestings of multiple queries, an equivalent method would be to use symbolic execution upon algebraic expressions representing the relational operations of the nested queries. Nesting is symbolically represented in such algebraic expressions as a function which appears in an outer nested query and has an argument including the relational operations of an inner nested block. The method for evaluating the function for the pair of outer and inner nested blocks defines a rule or pattern which is applied or matched to the algebraic expression, and when a match is found, the substitution or change to the expression that should be performed to remove the function. *Hardy*, supra, discloses further details regarding symbolic execution.

The un-nested query blocks themselves specify an order of evaluation that is not necessarily the best order due to the commutative and associative properties of joins, and similar properties applicable to some operations between joins and outer-joins. *Dayal*, supra, exploits these properties to generalize Ganski's solution for queries with more than 2 blocks. A linear query with multiple blocks gives rise to a 'linear J/OJ expression' where each instance of an operator is either a join or an outer join. A general linear J/OJ expression would like:

R J/OJ S J/OJ T J/OJ U J/0J . . .

Relation R is associated with the outermost block, relation S with the next inner block, and so on. An outer join is required if there is a COUNT between the respective blocks. In all other cases (AVG, MAX, MIN, SUM), only a join is required. The joins and outer joins are evaluated using the appropriate predicates.

Since joins and outer joins do not commute with each other in general, a legal order may be obtained by computing all the joins first and then computing the outer joins in a left to right order (top to bottom, if you like) (Dayal, supra.) For example, the expression R OJ S J T J U OJ V J W can be legally evaluated as ((R OJ (S J T J U)) OJ (V J W)). Since the joins may be evaluated in any order, the least expensive join order is selected for joining relations S, T and U.

Consider the following three block linear query:

```
SELECT R.a
FROM R
WHERE R.b OP1 (SELECT COUNT (S.*)
    FROM S
    WHERE R.c OP2 S.c
    AND S.d OP3 (SELECT COUNT (T.*)
        FROM T
        WHERE S.e OP4 T.e
        AND R.f OP5 T.f))
```

The corresponding linear expression is R OJ S OJ T and hence a legal order is (R OJ S) OJ T. The result is obtained by executing the following two queries:

Query 1:

$TEMP_1$ (#, a, b, *)=
SELECT R.#, R.a, R.b, S.*
FROM R, S, T
WHERE (R OJ S) OJ T
GROUP BY R.#, S.#
HAVING S.d OP_3 COUNT(T.*)

Query 2

SELECT $TEMP_1$.a
FROM $TEMP_1$
GROUP BY $TEMP_1$.#
HAVING $TEMP_1$.b OP_1COUNT($TEMP_1$.*)

The outer join predicates are implicit in Query 1. The predicate for R OJ S is (R.c OP_2 S.c), while the predicate for the second outer join with T is (S.e OP_4 T.e and R.f OP_5 T.f). Tuples from Query 1 may be pipelined into Query 2. Notice that if the query has d blocks, the total number of joins and outer joins will be (d-1). These will be followed by (d-1) groupby-having operations.

Although a valid J/OJ ordering is obtained by performing all the joins first, followed by the outer joins from left to right, it is sometimes possible to change this order. Consider, for example, the following query:

```
SELECT R.a
FROM R
WHERE R.b OP1 (SELECT COUNT (S.*)
      FROM S
      AND S.d OP2 (SELECT MAX (T.d)
        FROM T
        WHERE R.f OP3 T.f))
```

The J/OJ expression for the above query is R **0J** (S J T). Since there is no correlation predicate between the S and T relations, a cartesian product must be performed to compute (S J T). The outer join is then performed using the predicate (R.f OP_3 T.f). However, for each (r, s) pair, where r ∈ R and s ∈ S, MAX (T.d) depends only on r. Hence, we can precompute MAX (T.d) associated with each tuple of R as follows:

$TEMP_1$ (#, a, b, max)=
SELECT R.#, R.a, R.b, MAX (T.d)
FROM R, T
WHERE R.f OP_3 T.f—0J
GROUP BY R. #

Notice that $|TEMP_1| = |R|$. Essentially, $TEMP_1$ has all the attributes of R required for further processing along with the MAX (T.d) associated with each tuple of R. MAX (T.d) can be computed in this fashion because it occurred in the last block. Any aggregate that does not occur in the last block depends on the results of the blocks below it and hence cannot be evaluated before the blocks below it are evaluated. Also, an outer join was performed between R and T even though we were computing MAX (T.d). This is because COUNT (S.*) indirectly depends on each tuple of R as R is referenced inside the third block which is nested within the second block. Hence, all tuples of R must be preserved. For a tuple of R with no joining tuples in T, the MAX value is set to a null value (^). (Any comparison where one or both of the operands is null (^) evaluates to unknown, which SQL regards as false for query evaluation purposes.) The original query is now re-written as follows:

```
SELECT TEMP1.a
FROM TEMP1
WHERE TEMP1.b OP1 (SELECT COUNT (S.*)
      FROM S
      WHERE S.d OP2 TEMP1.max)
```

We now have a correlation predicate between $TEMP_1$ and S, thus avoiding a cartesian product. Similar ideas were presented in *Dayal*, supra, in his section titled "Positioning G-Agg Operations". In that section, Dayal presents rules for computing aggregates before G-joins.

In any case, the above example shows that it is possible to precompute the bottom-most aggregate (BMA) if the number of outer relations referenced in the last block have already been joined. Although in this example the bottom-most aggregate depended only on one outer relation, a further example is presented below where the bottom-most aggregate depends on more than one relation.

It is also possible to evaluate pipelined queries in a strictly top-down order, performing the joins and outer joins in the order they occur. Evaluation in a top-down manner may permit the method of FIG. **12** to be applied to a larger number of contiguous blocks at the end of the query. However, care must be taken to ensure that any join that is present just below an outer join is also evaluated as an outer join. Consider the following example:

```
SELECT R.a
FROM R
WHERE R.b OP1 (SELECT COUNT (S.*)
    FROM S
    WHERE R.c OP2 S.c
    AND S.d OP3 (SELECT MAX (T.d)
        FROM T
        WHERE S.e OP4 T.e
        AND R.f OP5 T.f))
```

The J/OJ expression is R OJ (S J T) . The join predicate between S and T is (S.e OP_4 T.e) and the outer join predicate is (R.c OP_2 S.c and R.f OP_5 T.f). Assume that the join between S and T is very expensive and should be possibly avoided. Could we evaluate (R OJ S) first? It turns out that we can indeed perform (R OJ S) first. However, some precautions/modifications are necessary.

It is clear that if an R tuple has no matching S tuples, the count associated with that R tuple is 0. As pointed out in *Murali*, supra, this R tuple may be routed to a higher node in the query tree so that it does not participate in the next join operation with T.

We thus need to consider only the join tuples of the form (r, s) from the outer join, where r ∈ R and s ∈ S. Let us focus our attention on a single tuple of R. When the join with T is evaluated using the predicate (S.e OP_4 T.e AND R.f OP_5 T.f), it is quite possible that none of these (r, s) tuples join with any tuples of T. In this case, the r tuple will be lost. However, if r.b equals 0, r is a result tuple and hence must be preserved. On the other hand, if some of the (r, s) tuples do join with some T tuples, it may so happen that after we do the groupby by (R. #, S. #) and evaluate MAX (T.d), none of the s.d values in the (r, s) tuples satisfy (s.d OP_3MAX (T.d)). We may be tempted to discard all the (r, s) groups. Again, if r.b equals 0, we need to preserve r.

We can preserve r if we perform the next join as an outer join. Also, the groupby operator must not discard any (r, s) group not satisfying (s.d OP_3MAX (T.d)). Instead, it must pass it on preserving the R portion of the tuple and nulling out the S portion of the tuple.

The same ideas were presented in *Murali*, supra, when un-nesting tree queries. Summarizing, if we encounter the expression R OJ S OJ T J U J V, we could evaluate it as ((R OJ S) OJ (T J U J V)). The above order corresponds to evaluating all the joins first. Another evaluation order could be ((((R OJ S) OJ T) OJ (U J V)). Now we have an outer join between T and U. Carrying this idea one step further, the above expression may also be evaluated as ((((R OJ S) OJ T) OJ U) OJ V).

By applying the methods of FIGS. **10**, **11** and **12** to pairs of blocks in a nested query having multiple blocks, it is possible to generate a multiplicity of alternative query graphs. The query optimizer may compute a cost associated with each query graph, and select the least expensive graph for execution.

In the following example, it will be assumed that the query optimizer operates on a kind of "join graph". An example of such a join graph is shown in FIG. **14**. The graph G=(V, E) consists of a set of vertices V and a set of directed edges E. There is a one-one correspondence between the blocks of the query and the elements of V.

Each element of V, except for the first vertex, is labeled either C (COUNT) or NC (Non COUNT). This labeling is clearly suggestive of the kind of aggregate (COUNT or Non COUNT) present in that block. The vertices are numbered 1 through d, where d is the current number of vertices in the graph. A directed edge is drawn from vertex i to j ($i<j$) if there is a correlation predicate in the jth block between the relations of blocks i and j.

The method of FIG. **12** may be applied to the last k blocks of a query ($0 \leqq k \leqq d$) if the last k vertices of the graph of the query satisfy the following properties:

(1) The in degree of every C vertex is at most 1;

(2) The edge incident on a C vertex corresponds to a neighbor predicate;

(3) All the edges incident with the last k vertices correspond to equi-join correlation predicates; and

(4) The relations in the first d-k blocks have already been joined.

The bottom-most aggregate may be precomputed if the in degree of the last vertex is at most 1.

The operations on the graph are as follows:

(1) When the relations of two or more blocks are joined, the corresponding vertices are collapsed into one vertex. The edges adjacent to these vertices are removed, while all the edges that connect these vertices to other vertices are preserved. Multiple edges are replaced by a single edge.

(2) Let d-**1** and d be the last two vertices in the graph. If the BMA is computed, the last vertex d is removed from the graph and the edge incident on d is connected to d-**1**.

Notice that we may be able to apply the method of FIG. **12** only after joining some relations. For example, we may apply the method of FIG. **12** to the last block after joining R and S in the query of Example **3**. This is because the predicate (R.f=T.f) becomes a neighbor predicate only after relations R and S are joined. Thus, the number of blocks for which we may apply the method of FIG. **12** can change dynamically. Similarly, the bottom-most aggregate may have originally depended on more than one outer relation, but, after these relations have been joined, the in degree of the last vertex will become 1. The bottom-most aggregate may be precomputed at this point.

When a series of consecutive m joins are encountered in a J/OJ expression, one may be tempted to evaluate all the joins using the cheapest order. However, the joins should be evaluated incrementally. In other words, the first i joins should be evaluated at a time, where $1 \leqq i \leqq m$. This ensures that we may be able to apply the method of FIG. **12** to a larger group of contiguous blocks at the end of the query.

Turning now to FIG. **15**, there is shown a flowchart of an un-nesting procedure that incorporates the above considerations for un-nesting multiple nested quires. In the first step **251** of FIG. **15**, execution branches depending on whether the bottom-most aggregate can be precomputed. It can be precomputed if the predicate of its node does not reference a relation in the relations list of any other nodes in which the bottom-most aggregate is nested. If so, then in step **252**, the bottom-most aggregate is computed and its corresponding node is removed from the query graph. Then, in step **253**, the query procedure of FIG. **15** is called recursively to process the amended query graph (G'). Execution then returns.

When step **251** finds that the bottom-most aggregate cannot be precomputed, then in step **254**, the nodes are identified to which Kim's method is applicable. These are the nodes that have aggregates other than COUNT, and which have equi-join predicates, and nodes with equi-join predicates and COUNTs known to always generate a count >0. Then, in step **255**, execution branches when Kim's method can be applied to all nodes. If so, in step **256**, Kim's method is applied from the bottom to the top of the query graph, and execution returns.

If Kim's method cannot be applied to all nodes, then in step **257**, the aggregate nodes to Which Kim's method cannot be applied are inspected to determine whether the method of FIG. **12** is applicable. The method of FIG. **12** is applicable to the count nodes with equi-join predicates that are neighbor predicates. If so, then in step **258**, the method of FIG. **12** or Ganski's method is chosen for these nodes depending upon a comparison of the computational cost of each respective method to each particular node. In step **259**, Ganski's method is chosen for the remaining aggregate nodes, because it is the only practical method. Then, in step **260**, possible execution orders are considered, beginning first with the possibility of joins followed by outer-joins. If joins are followed by outer-joins, then in step **261**, for i=1 to m, the first i joins are evaluated using the least costly join order. Then, in step **262**, the un-nesting procedure of FIGS. **15** and **16** is recursively called for the new graph including the remaining nodes, and execution returns.

If step **260** determines that there is not a sequence of joins followed by outer-joins, then execution branches to step **263**. In step **263**, execution returns if there is not a sequence of joins within outer joins. If so, then in step **264**, for i=1 to m, the first i joins are evaluated using the least costly join order. Then, in step **265**, the un-nesting procedure of FIGS. **15** and **16** is called recursively to operate upon the graph (G') of the remaining nodes. Then, in step **266**, the first outer join is evaluated, and the first join is replaced by an outer join. Then, in step **267**, the un-nesting procedure of FIGS. **15** and **16** is called recursively with the graph (G') including the remaining nodes. Execution then returns.

Consider the application of the procedure of FIGS. **15** and **16** to the following nested query, represented by the join graph of FIG. **14**:

```
SELECT R.a
FROM R
WHERE R.b OP1
    (SELECT COUNT (S.*)
    FROM S
    WHERE R.c = S.c
    AND S.d OP2
      (SELECT AVG (T.d)
      FROM T
      WHERE S.e = T.e
      AND T.g OP3
        (SELECT SUM (U.g)
        FROM U
        WHERE S.h = U.h
        AND T.i = U.i)))
```

The J/OJ expression is R - - - OJ - - - S - - - J - - - T - - - J - - - U. The alternative query plans shown in FIGS. **17** to **22** are possible.

In FIG. **17**, Kim's method and the method of FIG. **12** are applied to blocks **2**, **3** and **4**.

In FIG. **18**, R and S are joined by Ganski's method, and Kim's method is applied to blocks **3** and **4**. Since the

outer join between R and S is performed before the join, the first join is now evaluated as an outer join.

In FIG. 19, R, S and T are joined by Ganski's method, and Kim's method is applied to block 4. Notice that both joins are now replaced by outer joins.

In FIG. 20, Ganski's method is used to replace all joins by outer joins, followed by three groupby operations.

In FIG. 21, Kim's method is used to join relations S, T and U first, followed by outer joins from Ganski's method.

In FIG. 22, Kim's method is used to join relations S, T and U first. Since the bottom-most aggregate depends only on relations S and T, the bottom-most aggregate is computed before the outer join with R.

Notice that it was important to evaluate the joins incrementally. Most of the outer join nodes in FIGS. 17 to 22 have two output edges. The vertical edge represents the anti-join tuples, while the other edge represents the join tuples. Similarly, the groupby-having nodes have two output edges. The vertical edge represents the groups that did not satisfy that condition in the having clause. These groups have certain portions nulled out. For example, in FIG. 20, groups flowing from the first groupby-having node to the topmost groupby-having node along the vertical edge are of the form (R, NULL). Also, FIGS. 18 to 22 have edges that route tuples to a node much higher in the tree than the immediate parent. As pointed out in *Murali*, supra, this is optional but leads to savings in message costs.

In view of the above, there has been described a relational database system including a query optimizer that has available to it an alternative method for un-nesting queries that include a COUNT aggregate. In many cases the alternative method provides a more efficient way of executing the un-nested queries. In other cases the alternative method permits alternative query plans to be generated which may result in a savings in the total cost of the query.

What is claimed is:

1. A method of operating a digital computer for un-nesting an inner query from an outer query, said inner query referencing a first relation also referenced in said outer query, said inner query including a first predicate joining said first relation to a second relation, said inner query also including a count aggregate, said outer query having a second predicate referencing said first relation and said inner query, said method comprising the steps of:

a) converting said inner query to a first un-nested query by removing said first predicate and modifying said count aggregate function to count over groups of distinct values of said second relation; and

b) converting said outer query to a second un-nested query receiving results of said inner query by modifying said second predicate so that said second predicate is applied to said first relation and said results for values of said first relation which are joined to said results by said first predicate and so that said second predicate is applied to said first relation and a value of zero for values of said first relation which are not joined to any of said results by said first predicate.

2. The method as claimed in claim 1, wherein said inner query and said outer query are expressed in a query language including alphanumeric names for said first and second relations and alphanumeric symbols for operators in said first and second predicates.

3. The method as claimed in claim 2, wherein said inner query includes a select statement including the words "SELECT" and "COUNT" and said first predicate is included in a where statement including the word "WHERE".

4. The method as claimed in claim 2, further comprising the step of parsing said inner and outer query to form a linked data structure including a first node representing said inner query and a second node representing said outer query.

5. The method as claimed in claim 4, wherein said steps of converting generate another linked data structure including a third node defining said first un-nested query and a fourth node defining said second un-nested query.

6. The method as claimed in claim 5, wherein said another linked data structure includes a series of nodes defining a sequence of join and outer-join operations including a consecutive series of m joins, and wherein the first i joins are evaluated at a time first for $i=1$, then for $i=2$, then for $i=3, \ldots$, and finally for $i=m$.

7. The method as claimed in claim 6, wherein said evaluating generates a respective linked data structure for each value of i representing an alternative query plan.

8. The method as claimed in claim 4, wherein said linked data structure is hierarchical and includes a multiplicity of nodes descendant from a root node, and said steps of converting are performed upon parent-child pairs of nodes when the child nodes are leaf nodes in said linked data structure.

9. The method as claimed in claim 1, further comprising the step of executing said second un-nested query by indexing said results of said first un-nested query with a first index and indexing values of said first relation with a second index, and for each value of said first relation that is indexed by said second index, sequencing said first index and iteratively testing said first predicate with said each value of said first relation and one of said results indexed by said first index, and when said testing finds that said first predicate is true, testing said second predicate with said each value of said first relation and the indexed one of said results and terminating said testing of said first predicate with said each value of said first relation, and when said testing has tested said first predicate for said each value of said first relation and all of said results without ever finding that said first predicate is true, applying said second predicate to said each value of said first relation and a value of zero.

10. A method of operating a digital computer for executing an outer query including a nested inner query, said inner query referencing a first relation also referenced in said outer query, said inner query including a first predicate joining said first relation to a second relation, said inner query also including a count aggregate, said outer query having a second predicate referencing said first relation and said inner query, said method comprising the steps of:

a) computing a third relation including values of said second relation by counting over groups of distinct values of said second relation to compute a count for each distinct value of said second relation; and

b) applying said first predicate to combinations of values of said first relation and the distinct values of said second relation, and for combinations of values of said first relation and distinct values of said sec-

ond relation which satisfy said first predicate, applying said second predicate to said first relation and the count of said third relation for each distinct value of said second relation, and for values of said first relation for which said first predicate is not satisfied in combination with any of said distinct values of said second relation, applying said second predicate to said first relation and a value of zero.

11. The method as claimed in claim **10**, wherein said inner query and said outer query are expressed in a query language including alphanumeric names for said first and second relations and alphanumeric symbols for operators in said first and second predicates.

12. The method as claimed in claim **11**, wherein said inner query includes a select statement including the words "SELECT" and "COUNT", and said first predicate is included in a where statement including the word "WHERE".

13. The method as claimed in claim **11**, further comprising the step of parsing said inner and outer query to form a linked data structure including a first node representing said inner query and a second node representing said outer query.

14. The method as claimed in claim **13**, further comprising the step of converting said linked data structure into another linked data structure including a third node defining a first un-nested query for generating said third relation and a fourth node defining a second un-nested query for receiving said third relation, and wherein said step (a) of computing is performed by executing said first un-nested query and said step (b) of applying is performed by executing said second un-nested query.

15. The method as claimed in claim **14**, wherein said another linked data structure includes a series of nodes defining a sequence of join and outer-join operations including a consecutive series of m joins, and wherein the first i joins are evaluated at a time first for $i=1$, then for $i=2$, then for $i=3, \ldots$, and finally for $i=m$.

16. The method as claimed in claim **14**, wherein said linked data structure formed by said parsing is hierarchical and includes a multiplicity of nodes descendant from a root node, and said step of converting is performed upon parent-child pairs of nodes when the child nodes are leaf nodes in said linked data structure formed by said parsing.

17. The method as claimed in claim **10**, wherein said step (b) of applying includes indexing said third relation with a first index and indexing all values of said first relation with a second index, and for each value of said first relation, sequencing said first index and iteratively testing said first predicate with said each value of said first relation and one of said values of said second relation indexed by said first index, and when said testing finds that said first predicate is true, testing said second predicate with said each value of said first relation and the count for the indexed one of said values of said second relation and terminating said testing of said first predicate with said each value of said first relation, and when said testing has tested said first predicate for said each value of said first relation and all of said values of said second relation from said third relation without ever finding that said first predicate is true, applying said second predicate to said each value of said first relation and a value of zero.

18. A database system comprising, in combination:

a memory having stored in it a relational database;

an input device for receiving a query from a user requesting data from said relational database;

an output device for transmitting to the user data from said relational database; and

a query processing system including means for parsing said query to generate a query graph in said memory representing said query, means for generating an optimized query from said query graph; and means for executing said optimized query;

wherein said means for generating an optimized query includes means for un-nesting an inner query from an outer query, said inner query referencing a first relation also referenced in said outer query, said inner query including a first predicate joining said first relation to a second relation, said inner query also including a count aggregate, said outer query having a second predicate referencing said first relation and said inner query, said means for generating including:

a) means for converting said inner query to a first un-nested query by removing said first predicate and modifying said count aggregate function to count over groups of distinct values of said second relation; and

b) means for converting said outer query to a second un-nested query receiving results of said inner query by modifying said second predicate so that said second predicate is applied to said first relation and said results for values of said first relation which are joined to said results by said first predicate and so that said second predicate is applied to said first relation and a value of zero for values of said first relation which are not joined to any of said results by said first predicate.

19. The relational database system as claimed in claim **18**, wherein said means for executing said optimized query includes means for executing said second un-nested query by indexing said results of said first un-nested query with a first index and indexing all values of said first relation with a second index, and for each value of said first relation that is indexed by said second index, sequencing said first index and iteratively testing said first predicate with said each value of said first relation and one of said results indexed by said first index, and when said testing finds that said first predicate is true, testing said second predicate with said each value of said first relation and the indexed one of said results and terminating said testing of said first predicate with said each value of said first relation, and when said testing has tested said first predicate for said each value of said first relation and all of said results without ever finding that said first predicate is true, applying said second predicate to said each value of said first relation and a value of zero.

20. The method as claimed in claim **18**, wherein said means for parsing includes means for parsing a series of a multiplicity of nested queries to generate a query graph including a multiplicity of linked nodes, said means for un-nesting includes means for generating another query graph including a series of a multiplicity of linked nodes representing un-nested queries each having either a join or an outer-join operation and together defining a sequence of join and outer-join operations including a consecutive series of m joins, and wherein said means for optimizing further includes means for generating a plurality of alternative query graphs by evaluating the first i joins at a time first for $i=1$, then for $i=2$, then for $i=3, \ldots$, and finally for $i=m$.

* * * * *

Appendix C: Related Proceedings

None. Appellant, the undersigned Attorney for Appellant, and the Assignee know of no applications on appeal that may directly affect or be directly affected by or have a bearing on the Board's decision in the pending appeal.